



---

# Newton Toolkit User's Guide

**Preliminary.**  
**Apple Confidential © Apple Computer, Inc. 1996**

 Apple Computer, Inc.  
© 1993 - 1996 Apple Computer, Inc.  
All rights reserved.

No part of this publication or the software described in it may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format. You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

Printed in the United States of America.

The Apple logo is a registered trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for licensed Newton platforms.

Apple Computer, Inc.  
1 Infinite Loop  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, APDA, AppleLink, AppleTalk, LaserWriter, Macintosh, MPW, Newton, PowerBook, and Power Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Finder, the light bulb logo, MessagePad, NewtonScript, Newton Toolkit, PowerBook Duo, ResEdit, and System 7 are trademarks of Apple Computer, Inc.

Adobe Illustrator and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Windows95, Windows NT3.5, and Windows 3.1 are registered trademarks of Microsoft Corporation.

Simultaneously published in the United States and Canada.

#### LIMITED WARRANTY ON MEDIA AND REPLACEMENT

If you discover physical defects in the manual or in the media on which a software product is distributed, APDA will replace the media or manual at no charge to you provided you return the item to be replaced with proof of purchase to APDA.

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

# Contents

Preface	About This Book	xxi
---------	-----------------	-----

---

Related Books	xxi
How to Use This Book	xxi
Conventions	xxiii
Developer Products and Support	xxv

Chapter 1	Installation and Setup	1-1
-----------	------------------------	-----

---

System Requirements	1-1
Installing NTK on the Development System	1-2
Installing the Toolkit Application on the Newton	1-3
Making the Physical Connection	1-4
Downloading the Toolkit Application	1-4
Testing an Inspector Connection	1-6
Troubleshooting	1-7

Chapter 2	Programming With the
Newton Toolkit	2-1

---

Terms and Concepts	2-1
The NTK Development Process	2-3

Chapter 3	
A Quick Tour of NTK	3-1

---

Starting Up NTK	3-2
-----------------	-----

Setting Up a New Project	3-2
Starting a Layout File and Adding It to the Project	3-6
Laying Out Application Elements	3-10
Customizing a View Template	3-11
Editing a Slot	3-11
Adding a Slot	3-13
Building and Downloading a Package	3-14
Adding a Linked Layout	3-16
Laying Out a Linked View	3-17
Linking in the Layout	3-19
Adding a Button That Displays the View	3-21
Defining Your Own Proto	3-23
Laying Out a Proto and Adding It to the Toolbar	3-23
Using Your Proto	3-29
Using the Inspector	3-30
Connecting the Inspector	3-31
Executing Commands	3-31
Looking at a Frame and a View	3-33
Making a Change in a Running Application	3-34

---

<b>Chapter 4</b>	<b>Managing and Building a Project</b>	<b>4-1</b>
------------------	--	------------

---

Setting Up a Project	4-1
Project File	4-2
Layout Files	4-3
Text Files	4-4
Bitmap and Sound Files	4-4
Package Files	4-5
Object Stream Files	4-5
Establishing Settings and Preferences	4-6
Application Settings	4-6
Package Settings	4-9
Project Settings	4-12
Output Settings	4-16

Output	4-17
Result	4-19
Toolkit Preferences	4-19
App Preferences	4-20
Packages Preferences	4-21
Connection	4-22
Build	4-22
Heaps Preferences	4-23
Layout Preferences	4-24
Browser Preferences	4-25
Browsers	4-26
Text Views Preferences	4-27
Building a Project	4-28
The Build Environment	4-29
Global Data File	4-29
Platform Files	4-30
Text Files	4-31
Constants and Variables	4-34
Compile-Time Functions	4-36
Defining Global Constants	4-37
Accessing Processed Templates	4-39
Accessing the Part Frame	4-40
Accessing Files That Aren't in the Project	4-41
Project-Build Function Summary	4-43
Build Options	4-43
Compiling Native Code	4-43
Embedding Debugging Information	4-44
Combining Objects	4-44
Profiling	4-45
Establishing a Local Language	4-46
Output Options	4-47
Build Sequences	4-52
Building a Project	4-52
Processing a Template	4-53
Error Messages	4-54

## Chapter 5      Laying Out and Editing Views      5-1

---

Laying Out Views	5-1
Drawing, Resizing, and Moving Views	5-4
Drawing a View	5-4
Resizing a View	5-7
Moving a View	5-8
Aligning Views	5-8
Ordering Views	5-10
Previewing	5-11
Naming and Declaring Views	5-13
Linking Multiple Layouts	5-14
Creating User Protos	5-16
Browsing and Editing Templates	5-16
Browsing Templates	5-16
Adding Slots	5-18
Editing Slots	5-20
Editing Text	5-23
Searching for Text in Files	5-25
Searching Template Files	5-25
Searching the Active Window	5-26
Finding Views in a Layout File	5-27
Adding Non-View Objects	5-28
Customizing the Text Editor	5-28

## Chapter 6      Debugging      6-1

---

Compatibility	6-2
The Inspector	6-2
Using the Inspector	6-5
Making an Inspector Connection	6-5
Retrieving Views	6-7
Displaying the View Hierarchy	6-8
Displaying Values in the Inspector Window	6-9
Examining a Binary Object	6-11

Breaking	6-11	
Examining the Program Stack	6-12	
Tracing the Flow of Execution	6-14	
Examining Memory Use	6-16	
Examining Drawing Efficiency	6-20	
Debugging Variables	6-21	
Debugging Functions	6-22	
Retrieving and Displaying Objects	6-23	
Using Break Loops	6-26	
Examining Memory Use	6-28	
Examining Drawing Efficiency	6-30	
Debugging Function Summary	6-31	
Retrieving and Displaying Objects	6-31	
Using Break Loops	6-31	
Examining Memory Use	6-31	
Examining Drawing Efficiency	6-31	
Newton Programming Problems and Tips	6-32	
Common Programming Problems	6-32	
Setting the Wrong Slot Value	6-32	
Failing to Set a Return Value	6-34	
Producing Memory Problems With Unused Frame References	6-34	
Generating Unexpected Comparison Results With nil Values	6-34	
Using nil in Expressions	6-36	
Writing to a Read-Only Object	6-36	
Text Is Not Drawing	6-38	
Problems with Printing and Communications	6-38	
Programming Tips for Debugging	6-39	
Using Global Variables to Examine Exceptions	6-39	
Maintaining View State	6-39	
Accessing the Parent of a View	6-40	

Chapter 7	Extended Debugging Functions	7-1
<hr/>		
Compatibility	7-2	
Installing the Extended Debugging Functions	7-2	
Using the Extended Debugging Functions	7-2	
Break Loops and Break Points	7-3	
Enabling Break Points	7-3	
Creating, Removing, and Disabling Break Points	7-4	
Making Break Points Conditional	7-5	
Entering a Break Loop	7-5	
NewtonScript Stacks	7-6	
Paths to Slots	7-7	
NewtonScript Byte Code	7-7	
Extended Debugging Functions Reference	7-9	
Adjusting the Debugging Environment	7-10	
Manipulating Break Points	7-10	
User-Defined Breakpoint Functions	7-13	
Stepping	7-15	
Accessing the Stack	7-16	
Retrieving Paths	7-20	
Disassembling	7-21	
Summary of Extended Debugging Functions	7-22	
Manipulating Break Points	7-22	
Stepping	7-23	
Accessing the Stack	7-23	
Retrieving Paths	7-23	
Disassembling	7-23	
Interpreter Instructions	7-23	
Stack Operations	7-25	
Program Flow	7-30	
While and Repeat/Until Loops	7-30	
For Loops	7-32	
Foreach Loops (Frame and Array Iterators)	7-34	
Exception Handling	7-36	
Calling and Returning Functions	7-37	
Primitive Functions	7-40	



## Chapter 8                      Tuning Performance                      8-1

---

Measuring Performance	8-1
Marking Functions for Profiling	8-2
Configuring the Compiler for Profiling	8-4
Configuring the Profiler on the Newton	8-6
Collecting Statistics	8-7
Interpreting a Profile	8-8
Compiling Functions for Speed	8-10
Declaring and Typing Variables	8-11
Stepping Through an Array	8-13
Handling Exceptions	8-13
Calling Other Functions	8-13
Calling Options	8-14
Timing Interactions	8-16
An Optimization Example	8-17
Profiling Native Functions	8-19

## Chapter 9                      NTK Commands                      9-1

---

File Menu	9-1
New Layout (Ctrl-N)	9-1
New Proto Template (Ctrl-T)	9-2
New Text File	9-2
Open (Ctrl-O)	9-2
Link Layout	9-2
Close (Ctrl-W)	9-3
Save (Ctrl-S)	9-3
Save As	9-3
Save All (Ctrl-M)	9-3
Revert	9-4
Print Setup	9-4
Print One	9-4
Print (Ctrl-P)	9-4
Exit	9-4

Recent File	9-4
Edit Menu	9-5
Undo (Ctrl-Z)	9-5
Redo (Ctrl-A)	9-5
Cut (Ctrl-X)	9-5
Copy (Ctrl-C)	9-5
Paste (Ctrl-V)	9-5
Clear (Delete)	9-6
Duplicate (Ctrl-D)	9-6
Shift Left	9-6
Shift Right	9-6
Select All (Ctrl-A)	9-6
Select Hierarchy	9-6
Select in Layout	9-7
Search (Ctrl-R)	9-7
Find (Ctrl-F)	9-8
Find Next (Ctrl-G)	9-8
Find Inherited	9-8
Newt Screen Shot	9-9
Toolkit Preferences	9-9
Project Menu	9-15
New Project	9-15
Open Project	9-15
Add Window	9-15
Add File	9-15
Remove File	9-15
Update Files	9-16
Build Package (Ctrl-1)	9-16
Download Package (Ctrl-2)	9-16
Export Package to Text	9-16
Install Toolkit App	9-17
Mark as Main Layout	9-17
Process Earlier (Ctrl-Up Arrow)	9-17
Process Later (Ctrl-Down Arrow)	9-17
Settings	9-17
Layout Menu	9-21

Layout Size	9-22
Autogrid On	9-22
Set Grid	9-22
Move To Front	9-23
Move Forward (Ctrl-Down Arrow)	9-23
Move To Back	9-23
Move Backward (Ctrl-Up Arrow)	9-23
Alignment	9-24
Align	9-25
Preview (Ctrl-Y)	9-25
Browser Menu	9-25
Template Info (Ctrl-I)	9-25
New Slot	9-26
Rename Slot	9-27
Templates By Type	9-27
Templates By Hierarchy	9-27
Slots By Name	9-28
Slots By Type	9-28
Show Slot Values	9-28
Apply (Ctrl-E)	9-28
Revert	9-28
Window Menu	9-29
Open Inspector	9-29
Connect Inspector (Ctrl-K)	9-29
New Browser (Ctrl-B)	9-29
Open Layout (Ctrl-L)	9-29
Cascade	9-30
Tile	9-30
Arrange Icons	9-30
Set Default Window Position	9-30
Help Menu	9-30
Index	9-30
Command Reference	9-30
Using Help	9-30
About Newton Toolkit	9-31

<b>Appendix A</b>	<b>Keyboard Text-Editing Commands</b>	<b>A-1</b>
	Setting the Insertion Point	A-1
	Selecting Text	A-3
	Manipulating Selected Text	A-4
	Deleting Text	A-5
<b>Appendix B</b>	<b>Keyboard Shortcuts</b>	<b>B-1</b>
<b>Appendix C</b>	<b>Custom Bitmaps and Sounds</b>	<b>5</b>
	Adding Bitmap and Sound Files to a Project	5
	Using Bitmap and Sound Files	5
	Opening and Closing Resource Files	5
	Using the Resource-Handling Functions	6
	Using Bitmaps	6
	Making a Bitmap From a 'BMP' File	7
	Using External Sound Files	8
	Custom Functions	8
	Retrieving Resources	8
	Summary of Custom Functions	11
	Getting Custom Data	11
<b>Appendix D</b>	<b>Specialized Slot Editors</b>	<b>D-1</b>
	Script Slots	D-1
	View Attributes	D-2
	viewBounds	D-2
	viewFlags	D-4
	viewFormat	D-4
	viewJustify	D-4

viewEffect	D-5
viewTransferMode	D-5
Specific Slots	D-5

---

Appendix E	Newton Debugging Applications	E-1
------------	-------------------------------	-----

---

Installing the Debugging Packages	E-1
HeapShow	E-2
About HeapShow	E-2
About Newton Memory Management	E-2
Using HeapShow	E-3
Statistics Display	E-4
Preferences	E-5
HeapShow Controls	E-8

---

Chapter 10	Glossary	GL-1
------------	----------	------

---



# List of Figures

Chapter 1	Installation and Setup	1-1
	<b>Table 1-1</b>	Hardware and software requirements 1-2
	<b>Figure 1-1</b>	The Newton Toolkit application icon 1-4
	<b>Figure 1-2</b>	Toolkit Preferences- 1-5
	<b>Figure 1-3</b>	The Toolkit application open on the Newton 1-7
Chapter 2	Programming With the Newton Toolkit	2-1
	<b>Figure 2-1</b>	The Newton application development process 2-4
Chapter 3	A Quick Tour of NTK	3-1
	<b>Figure 3-1</b>	Layout window and toolbar 3-7
	<b>Figure 3-2</b>	A browser window 3-12
Chapter 4	Managing and Building a Project	4-1
	<b>Figure 4-1</b>	The project window 4-2
	<b>Figure 4-2</b>	Settings-Application 4-7
	<b>Figure 4-3</b>	Settings-Package 4-10
	<b>Figure 4-4</b>	Settings-Project 4-13
	<b>Figure 4-5</b>	Settings-Output 4-17
	<b>Figure 4-6</b>	Toolkit Preferences-App 4-20
	<b>Figure 4-7</b>	Toolkit Preferences-Packages 4-21
	<b>Figure 4-8</b>	Toolkit Preferences-Heaps 4-23
	<b>Figure 4-9</b>	Toolkit Preferences-Layout 4-24
	<b>Figure 4-10</b>	Toolkit Preferences-Browsers 4-25
	<b>Figure 4-11</b>	The Text Style dialog box 4-26

<b>Figure 4-12</b>	Toolkit Preferences-Text Views	4-28
<b>Table 4-1</b>	Build constants defined by NTK	4-34
<b>Figure 4-13</b>	Output Settings	4-48
<b>Figure 4-14</b>	Custom part settings	4-52

## Chapter 5

### Laying Out and Editing Views 5-1

---

<b>Figure 5-1</b>	Layout window and toolbar	5-2
<b>Figure 5-2</b>	A layout window with the layout view and one child view in place	5-6
<b>Figure 5-3</b>	The Alignment dialog box	5-9
<b>Figure 5-4</b>	The layout window in layout and preview modes	5-12
<b>Figure 5-5</b>	The Template Info dialog box, for naming and declaring views	5-13
<b>Figure 5-6</b>	Declaring views across linked layout files	5-15
<b>Figure 5-7</b>	A browser window with the view flags slot open for editing	5-17
<b>Figure 5-8</b>	The New Slot dialog box	5-19
<b>Figure 5-9</b>	The Editor drop list in the New Slot dialog box	5-20
<b>Figure 5-10</b>	Initial contents of evaluate, script, and text slots	5-21
<b>Figure 5-11</b>	The number, Boolean, rectangle, and picture slot editors	5-22
<b>Figure 5-12</b>	The Inspector window with a help message displayed	5-24
<b>Figure 5-13</b>	The Search dialog box	5-25
<b>Figure 5-14</b>	The dialog for searching with Find	5-27

## Chapter 6

### Debugging 6-1

---

<b>Figure 6-1</b>	Inspector window	6-3
<b>Figure 6-2</b>	The debugging cycle	6-4
<b>Figure 6-3</b>	Inspector controls	6-5
<b>Figure 6-4</b>	The DV display	6-8
<b>Figure 6-5</b>	A TrueSize display	6-16
<b>Figure 6-6</b>	A TrueSize display with object list	6-17
<b>Figure 6-7</b>	The TrueSize summary and result frame	6-18



<b>Figure 6-8</b>	A TrueSize listing of references	6-19
<b>Figure 6-9</b>	TrueSize measurements over time	6-20
<b>Table 6-1</b>	Debugging variables	6-21
<b>Table 6-2</b>	Exception handling global variables	6-39

## Chapter 7

## Extended Debugging Functions 7-1

---

## Chapter 8

## Tuning Performance 8-1

---

<b>Figure 7-1</b>	A performance profile	8-4
<b>Figure 7-2</b>	The Project Settings dialog box	8-5
<b>Figure 7-3</b>	Profile Control on the Newton	8-6
<b>Figure 7-4</b>	Profiler Info	8-6
<b>Figure 7-5</b>	Profiler Settings on the Newton	8-7
<b>Figure 7-6</b>	A performance profile	8-8
<b>Table 7-1</b>	Utility functions optimized for calling as global functions from a native function	8-14
<b>Table 7-2</b>	Function call operations	8-17
<b>Figure 7-7</b>	A profile of a native function calling another native function, without native-function profiling	8-20
<b>Figure 7-8</b>	A profile of a native function calling another native function, with native-function profiling	8-21

## Chapter 9

## NTK Commands 9-1

---

<b>Figure8-1</b>	The dialog for searching with Search	9-7
<b>Figure0-1</b>	The dialog for searching with Find	9-8
<b>Figure8-2</b>	The App preferences of the Toolkit Preferences dialog box	9-9
<b>Figure8-3</b>	The Layout preferences of the Toolkit Preferences dialog box	9-10
<b>Figure8-4</b>	The Browsers preferences of the Toolkit Preferences dialog box	9-11
<b>Figure8-5</b>	The Text Views preferences of the Toolkit Preferences dialog box	9-12
<b>Figure8-6</b>	The Packages preferences of the Toolkit Preferences dialog box	9-13

<b>Figure8-7</b>	The Heaps preferences of the Toolkit Preferences dialog box	9-14
<b>Figure8-8</b>	The Application Settings panel of the Settings dialog box	9-18
<b>Figure8-9</b>	The Package Settings panel of the Settings dialog box	9-19
<b>Figure8-10</b>	The Project Settings panel of the Settings dialog box	9-20
<b>Figure8-11</b>	The Output Settings panel of the Settings dialog box	9-21
<b>Figure8-12</b>	The Layout Size dialog box	9-22
<b>Figure8-13</b>	The Set Grid dialog box	9-23
<b>Figure8-14</b>	The Alignment dialog box	9-24
<b>Figure8-15</b>	The alignment buttons on the palette	9-24
<b>Figure8-16</b>	The Template Info dialog box, for naming and declaring views	9-26
<b>Figure8-17</b>	The New Slot dialog box	9-26
<b>Figure8-18</b>	The Rename Slot dialog box	9-27

## Appendix A

### Keyboard Text-Editing Commands A-1

---

<b>Table A-1</b>	Moving the insertion point	A-2
<b>Table A-2</b>	Selecting text with keyboard commands	A-3
<b>Table A-3</b>	Manipulating selected text	A-4
<b>Table A-4</b>	Deleting text with keyboard commands	A-5

## Appendix B

### Keyboard Shortcuts B-1

---

<b>Table B-1</b>	Keyboard equivalents to menu items	B-1
<b>Table B-2</b>	Keyboard commands that affect the hierarchy	B-2

## Appendix C

### Custom Bitmaps and Sounds 5

---

<b>Figure C-1</b>	Adding a named 'BMP' file to a picture slot	7
-------------------	---	---

## Appendix D

### Specialized Slot Editors D-1

---

**Table D-1**      Meaning of viewBounds fields for horizontal justification      D-3

**Table D-2**      Meaning of viewBounds fields for vertical justification      D-3

## Appendix E

### Newton Debugging Applications E-1

---

**Figure D-1**      The HeapShow icon      E-3

**Figure D-2**      The default HeapShow display      E-4

**Figure D-3**      Numerical data versus fragmentation graphics      E-5

**Figure D-4**      HeapShow Preferences      E-6

**Figure D-5**      Sizing the reserve pointers heap or a newly created heap      E-7

**Figure D-6**      Check Interval options      E-8

**Figure D-7**      The HeapShow controls      E-8

**Figure D-8**      Heap fragmentation graphics      E-9

## Chapter 10

### Glossary GL-1

---



# About This Book

---

This book documents release 1.6 of the Newton Toolkit (NTK), an integrated environment for developing applications that run on the Newton family of personal digital assistants (PDAs).

## Related Books

---

This book is one of two shipped with NTK. Its companion is *The NewtonScript Programming Language*, which documents the language you use for programming in NTK.

You also use this book in conjunction with the *Newton Programmer's Guide*, a two-volume set that explains how to write Newton programs and describes the system software routines you use in your programs.

If you're using NTK to build on-line books, you need the *Newton Book Maker User's Guide*, which is shipped with the Book Maker software.

## How to Use This Book

---

This book is both an introduction and a reference guide to NTK. You use this book to learn the basics of NTK before you can begin using the other books in the Newton documentation suite. Later, you use this book to learn about testing and debugging your software.

You must read a few parts of this book carefully; other parts you can skim at the outset and come back to later. This book contains nine chapters:

- Chapter 1, “Installation and Setup.” Follow the instructions in this chapter to install the Newton Toolkit on the development system and on a Newton and to set up and test a connection to the Newton.
- Chapter 2, “Programming With the Newton Toolkit.” Read this chapter for an introduction to Newton programming terminology, an overview of the Newton development process, and a description of the basic components of the Newton Toolkit.
- Chapter 3, “A Quick Tour of NTK.” If you want a hands-on introduction to NTK, follow this tutorial to code, build, and download a simple Newton application.
- Chapter 4, “Managing and Building a Project.” Read the introduction to this chapter to learn how you organize a software project in NTK. Skim the rest of the chapter, and then use it as a reference when you’re actually setting up, coding, and building your software.
- Chapter 5, “Laying Out and Editing Views.” Skim this chapter to learn how you can use the graphical editor and the browser to lay out and code your software. Use it as a reference when you’re using the tools.
- Chapter 6, “Debugging.” Read the first part of this chapter to learn about the NTK debugging window—the Inspector—and the functions you use to examine an application under development. Skim the rest and use it as a reference when you’re using the tools.
- Chapter 7, “Extended Debugging Functions.” Read this chapter if you’re using the extended debugging functions, which let you look more closely at an application under development.
- Chapter 8, “Tuning Performance.” Read the appropriate parts of this chapter when you’re ready to use the NTK profiler to

collect performance statistics or the native compiler to speed up execution of selected functions.

- Chapter 9, “NTK Commands.” Use this chapter for reference.

This book also contains a number of appendices:

- Appendix A, “Keyboard Text-Editing Commands,” lists the keyboard commands you can use to manipulate text in NTK.
- Appendix B, “Keyboard Shortcuts,” lists the Command-key equivalents to NTK menu items and other keyboard shortcuts.
- Appendix C, “Custom Bitmaps and Sounds,” describes the functions you use to manipulate bitmap and sound files directly.
- Appendix D, “Specialized Slot Editors,” lists the special-purpose NTK slot editors.
- Appendix E, “Newton Debugging Applications,” lists the small Newton debugging functions shipped with NTK and documents the HeapShow application, which displays Newton memory statistics.

## Conventions

---

This book uses the following font and syntax conventions:

<i>Courier</i>	The Courier font represents material that is typed exactly as shown. Code listings, code snippets, and special identifiers in the text such as predefined system frame names, slot names, function names, method names, symbols, and constants are shown in the Courier typeface to distinguish them from regular body text.
<i>italics</i>	Text in italics represents replaceable elements, such as function parameters, which you must replace with your own values.

## P R E F A C E

<b>boldface</b>	Key terms and concepts are printed in boldface where they're defined.
...	An ellipsis in a syntax description means that the preceding element can be repeated one or more times.  An ellipsis in a code example represents code not shown.
[ ]	Square brackets enclose optional elements in syntax descriptions.



## Developer Products and Support

---

APDA is Apple's worldwide source for a large number of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Every four months, customers receive the *APDA Tools Catalog* featuring current versions of Apple's development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most products. APDA offers convenient payment and shipping options including site licensing.

To order product or to request a complimentary copy of the *APDA Tools Catalog*:

APDA

Apple Computer, Inc.

P.O. Box 319

Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
-----------	---

Fax	716-871-6511
-----	--------------

AppleLink	APDA
-----------	------

America Online	APDA
----------------	------

CompuServe	76666,2405
------------	------------

Internet	APDA@applelink.apple.com
----------	--------------------------

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

# P R E F A C E

# Installation and Setup

---

The Newton Toolkit (NTK) runs on various development systems in 32-bit mode with a minimum of 8 MB of RAM. Its companion application, the Toolkit App, runs on a Newton Personal Digital Assistant (PDA).

This chapter describes how to

- install NTK on a development system
- download the Toolkit application to a Newton PDA
- establish a debugging connection between the development system and the PDA
- troubleshoot installation and setup

## System Requirements

---

Table 1-1 lists NTK's hardware and software requirements.

## Installation and Setup

**Table 1-1** Hardware and software requirements

	<b>Recommended</b>	<b>Minimum</b>
<b>Model</b>	Pentium-based processor	25 Mhz 486-based processor
<b>Operating Systems</b>		
Windows NT 3.5	24 MB	16 MB
Windows 3.1 with Win32s extension	16 MB	8 MB
Windows 95	16 MB	8 MB

## Installing NTK on the Development System

---

The Newton Toolkit is shipped with an installation script, which you run from the distribution disk.

1. Exit any other applications that are running on the development system.
2. Insert the disk *Newton Toolkit Installer*.
3. Double-click the installation instructions and read them to learn what your options are and how to adjust the installation script to your needs.
4. From the File Manager, double-click on the Setup.bat file.  
This file will ask you for the drive on which you wish to install NTK. The batch file will create a directory named WinNTK on the selected drive and copy the necessary files into said directory. No other directories on your drive will be touched.
5. Using the Fonts control Panel, install the font Espy.fon into your system. This file can be found in the WinNTK directory just created in the preceding step.
6. In the File Manager, double-click on NTK.EXE to launch NTK.

## Installation and Setup

The Newton Toolkit directory contains the NTK application and various support tools:

- The Toolkit.pkg, an NTK companion application that runs on a Newton. Instructions for installing the Toolkit application appear in the next section.
- The EditCmds file, which is used by NTK and which must remain in the same directory as NTK.
- The MsgPad.txt file, contained in the Platfrms directory, which contains a list of compile-time constants and functions available in NTK. This file is for your information only and can be put anywhere.
- The Platfrms directory, which at this release contains platform files named MsgPad.ptf and Newton20.ptf and definition files for each.  
Platform files contain data specific to a Newton platform. The Platfrms directory must remain in the same directory as NTK. The definitions files contain lists of compile-time constants and functions available when you're using each platform. These files are for your information only and can be put anywhere.
- The Espy.fon file, which contains the fonts used by NTK. The Espy.fon file is installed as other fonts in your system—using the Fonts Control Panel.
- The Newton Package Installer, a stand-alone utility that downloads packages to a Newton. You are free to ship this utility as an installation tool for your customers.
- The Newton Debugging Tools directory, which contains a collection of debugging software. This software is described in Chapter 7, "Extended Debugging Functions," and Appendix E, "Newton Debugging Applications."
- Release notes for the Newton Toolkit and the Platform files. These files are for your information only and can be put anywhere.

## Installing the Toolkit Application on the Newton

---

You can install the Toolkit application over a direct serial connection. Once you've installed the Toolkit application, it manages subsequent installation of

## Installation and Setup

NTK packages and supports the Inspector, a debugging window that lets you examine software running on the Newton device.

## Making the Physical Connection

---

You can connect your development system directly to a Newton device with a serial cable. Development systems have differing serial port connections, so make sure to verify the necessary serial port settings before making the connection.

You connect the cable between the serial connector on the Newton device and one of the serial ports on the development system. You can any serial port; NTK is configured to use the serial port by default.

### Note

Serial ports come in two sizes—9-pin or 25-pin. On some systems, a serial port may be labeled COM1, COM2, COM3, or COM4. ♦

## Downloading the Toolkit Application

---

This section describes how to install the Toolkit application on a Newton, using a serial cable connection to one of the built-in ports on the development system.

1. On the development system, start NTK by double-clicking the Newton Toolkit application icon.

---

**Figure 1-1** The Newton Toolkit application icon

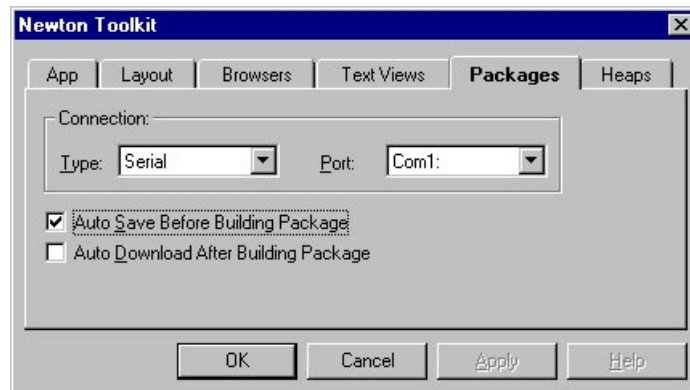


## Installation and Setup

The development system displays the open-file dialog box, for opening an NTK project.

2. Dismiss the dialog box by clicking Cancel.
3. Choose Toolkit Preferences from the Edit menu. The system displays the default Toolkit Preferences-App dialog box.
4. Click the tab labeled to view the Toolkit Preferences-Packages dialog box as shown in Figure 1-2.

**Figure 1-2** Toolkit Preferences-



5. Set the Connection Type to Serial.
6. Set the Port, depending on your configuration. For example, choose COM1 as the connection port if that is the chosen port on your development system.
7. Click OK.
8. Choose Install Toolkit App from the Project menu.  
NTK prompts you to initiate the connection on the Newton.

## Installation and Setup

9. On the Newton, tap the Connection application to open it. The exact appearance of the Connection application depends on what version you're using.
10. Verify that the connection setting matches your configuration.
11. Tap Connect.  
The Newton reports that the Connection application is waiting for a response. In a few seconds, the dialog disappears, and a toolbox icon labeled Toolkit appears in the Extras drawer.  
NTK Toolkit application installation is complete.

## Testing an Inspector Connection

---

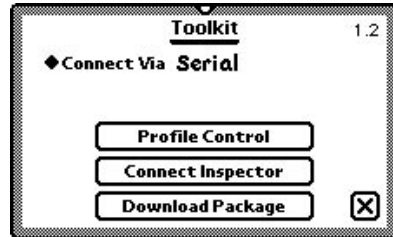
An Inspector connection lets you issue commands directly to the Newton device from a window on the development system. You can only use the Inspector over a serial line.

These instructions assume that you've already set the NTK connection preferences as described in "Downloading the Toolkit Application" beginning on page 1-4

1. On the development system, choose Connect Inspector from the Window menu.  
NTK prompts you to initiate the connection on the Newton.
2. On the Newton, open the Toolkit application by tapping its icon in the Extras drawer. Figure 1-3 illustrates the open Toolkit application.



## Installation and Setup

**Figure 1-3** The Toolkit application open on the Newton

The Profile Control button appears only if the attached Newton supports profiling.

3. Verify that Serial is selected, otherwise, change the selection.
4. Tap Connect Inspector.

The development system and the Newton establish communication, and the development system displays the Inspector window.

5. In the Inspector window, type these characters without pressing Return.

1/5

6. Press Enter.

The Inspector displays the value of the statement.

#440D2F1      0.200000

## Troubleshooting

If you have trouble launching NTK, try these troubleshooting strategies:

- Verify that the development system is using Windows 95, Windows NT 3.5, or Windows 3.1 with Win32s extensions as its operating system.

## Installation and Setup

- Verify that the directory containing the Newton Toolkit application also contains a Platforms directory that contains the MsgPad.ptf or Newton20.ptf file.
- Read the release notes.

If you have trouble downloading the Toolkit application or making an Inspector connection, try these troubleshooting strategies:

- Verify that you're using the appropriate serial cable and port.
- Reset the Newton.
- If the Newton device has little free space, remove some software.
- Read the release notes.

# Programming With the Newton Toolkit

---

The Newton interface is a graphical one, in which the user manipulates elements on the screen to accomplish a wide range of tasks.

The Newton Toolkit is an integrated environment tailored to the graphical nature of the Newton environment. This chapter introduces the concepts and terminology used in Newton programming and outlines the software development process.

## Terms and Concepts

---

**Views** are the basic building blocks of most applications. The individual items on the Newton screen—radio buttons, for example—are all views, and there may be views that are not visible.

You lay out views using NTK's graphical editor. When you draw a view, NTK creates a **template**, that is, a data object that describes how the view

## Programming With the Newton Toolkit

will look and act on the Newton. You build your application from a collection of templates that describe the application's elements.

A template is a **frame**, the basic data structure in NewtonScript. A frame is an object containing a collection of named data references called **slots**. You define a view's characteristics and behavior by specifying the contents of the slots in its template.

You write the code that controls the behavior of a view in NewtonScript, an object-oriented language developed for the Newton. NewtonScript is described in *The NewtonScript Programming Language*.

Views are created from templates when your application executes on the Newton. The process of making an object, such as a view, at run time, from a template, is called **instantiation**.

A view has two parts: the visual object you see on the screen, and a frame in memory containing transient data used at run time. This frame is sometimes called the view frame.

Applications can also include non-graphical components, such as communication services, that have no visible manifestations. Like views, these objects are described by templates and are instantiated at run time into a frame that exists in working RAM.

Newton applications are stored on ROM-based PCMCIA cards or in a protected part of the Newton memory. The Newton does not copy an application (in this case, a collection of templates) into working RAM when executing it. Therefore, templates are read-only objects. Views are their dynamic, writable counterparts.

When the Newton instantiates a view, it creates a view frame in working RAM. The view frame contains a pointer to the template. Information is read from the template as needed. If a value changes at run time, a slot is added to the view frame, and the new value is stored there. This memory-use strategy allows applications to use relatively small amounts of working RAM.

This architecture also makes available to your application all templates built into the Newton ROM. When you use a view template from the NTK toolbar (described in Chapter 5, "Laying Out and Editing Views"), your application doesn't have to contain the full template. Instead, NTK references the

## Programming With the Newton Toolkit

templates in the Newton ROM and places only your modifications in the application.

The *Newton Programmer's Guide* contains a full description of the Newton view system and the templates and functions you use when programming a Newton application.

## The NTK Development Process

---

You manage an application under development as an NTK **project**, that is, the collected files and specifications NTK needs to build a data package that can be downloaded to and executed on the Newton. The section “Setting Up a Project” beginning on page 4-1 describes how you organize a project in NTK.

You lay out an application's views with NTK's graphical editor and a toolbar of view templates. The graphical editor creates **layout files**, that is, files containing the templates that describe the application's views. The section “Laying Out Views” beginning on page 5-1 describes the graphical editor and toolbar.

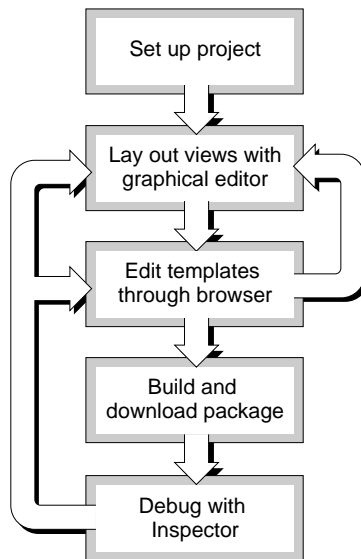
You use the NTK **browser** to search through and edit the templates in a layout file. The section “Browsing and Editing Templates” beginning on page 5-16 describes the NTK browser.

Once you've programmed your application, you use NTK to build a **package**, that is, a data object that can be installed on the Newton. The section “Building a Project” beginning on page 4-28 describes the build cycle. You also use NTK to download the package to the Newton.

You can study and alter your application while it's running with the Inspector, NTK's interactive debugger. Chapter 6, “Debugging,” describes the Inspector.

Figure 2-1 illustrates the application development process.

## Programming With the Newton Toolkit

**Figure 2-1** The Newton application development process

# A Quick Tour of NTK

---

This chapter introduces the major components of NTK and illustrates the Newton application development process.

You can follow this tutorial to lay out, build, download, and examine a simple application. The tutorial illustrates

- setting up an application project
- laying out the application's visual interface
- coding the application
- building an application package and downloading it to a Newton
- inspecting the application while it's running.

**Note**

This tutorial assumes that you're running NTK on a development system with a physical connection to a Newton device, as described in Chapter 1, "Installation and Setup." ♦

The following three chapters, "Managing and Building a Project," "Laying Out and Editing Views," and "Debugging," describe the primary

## A Quick Tour of NTK

components of NTK. You might want to read those chapters in conjunction with proceeding through the tutorial.

## Starting Up NTK

---

1. If NTK is not already running, double-click the Newton Toolkit application icon to start NTK.



NTK

NTK displays its startup screen followed by a file-open dialog box.

2. For this tutorial, click Cancel, because you're going to create a new project from scratch, not open an existing project.

## Setting Up a New Project

---

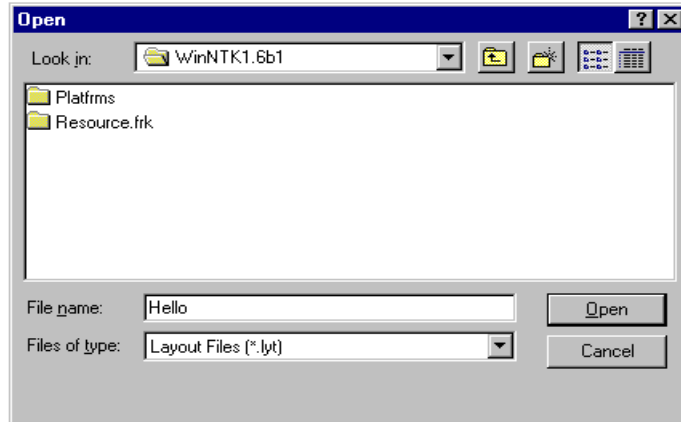
You build a Newton application from a collection of source files, which you coordinate through a **project file**. The first step in creating an application is to start a project file.

1. Choose New Project from the Project menu.  
NTK presents the standard file-save dialog box.



## A Quick Tour of NTK

2. Create a new directory named Hello, and change the project name from Project1 to Hello.



This tutorial appends .NTK to the project filename to distinguish it from other files stored in the same directory.

3. Click Save.

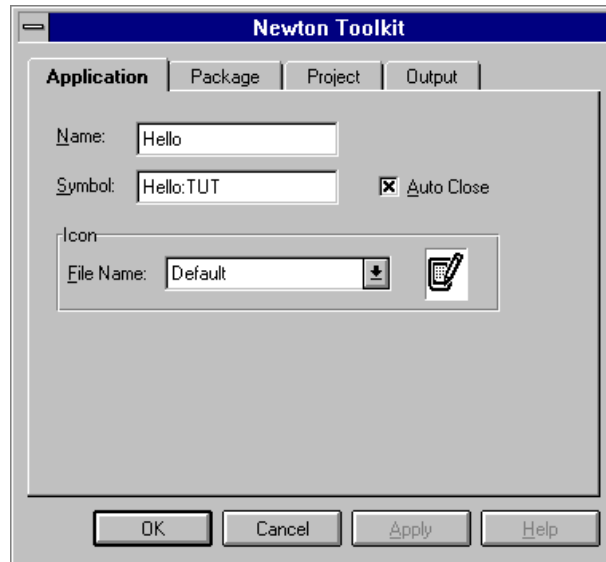
NTK displays the blank project window.



4. Choose Settings from the Project menu.

## A Quick Tour of NTK

NTK displays the application settings.

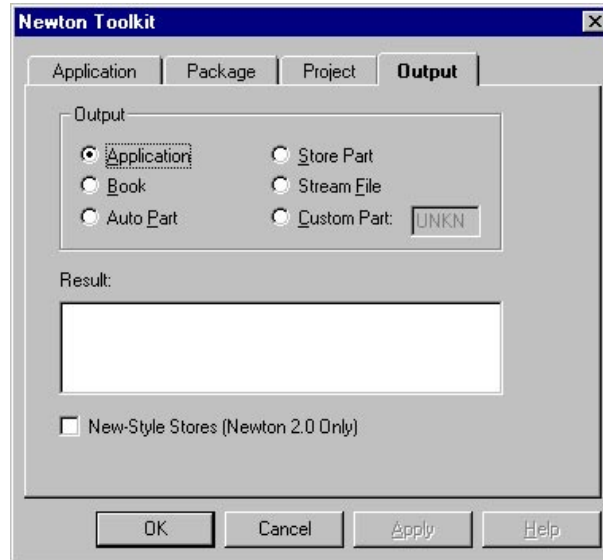


For this tutorial, change the Symbol to Hello:TUT, but don't click OK just yet.

Click the tab labeled Output.

## A Quick Tour of NTK

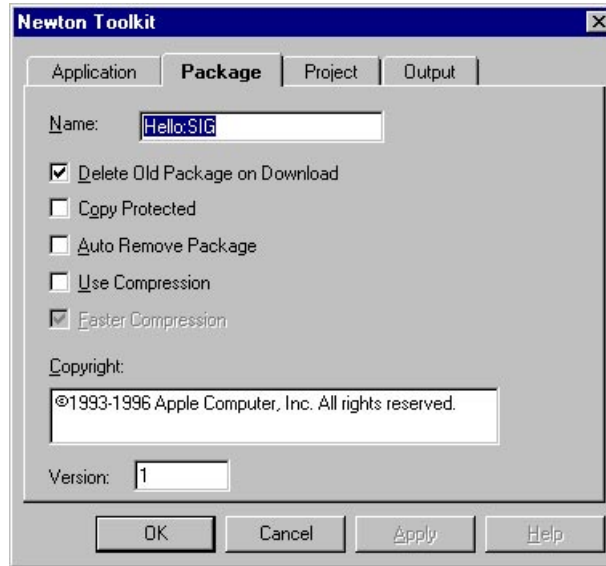
NTK displays the output settings, which determine the basic nature of the software—whether you’re building an application or a book, for example—and a few specifics about some software types.



5. Click the tab labeled Package.

## A Quick Tour of NTK

NTK displays the package settings, which specify the characteristics of an NTK package.



6. Change the package name to Hello:TUT and then click OK.

## Starting a Layout File and Adding It to the Project

---

NTK provides a graphical editor for arranging the visual elements of your application.

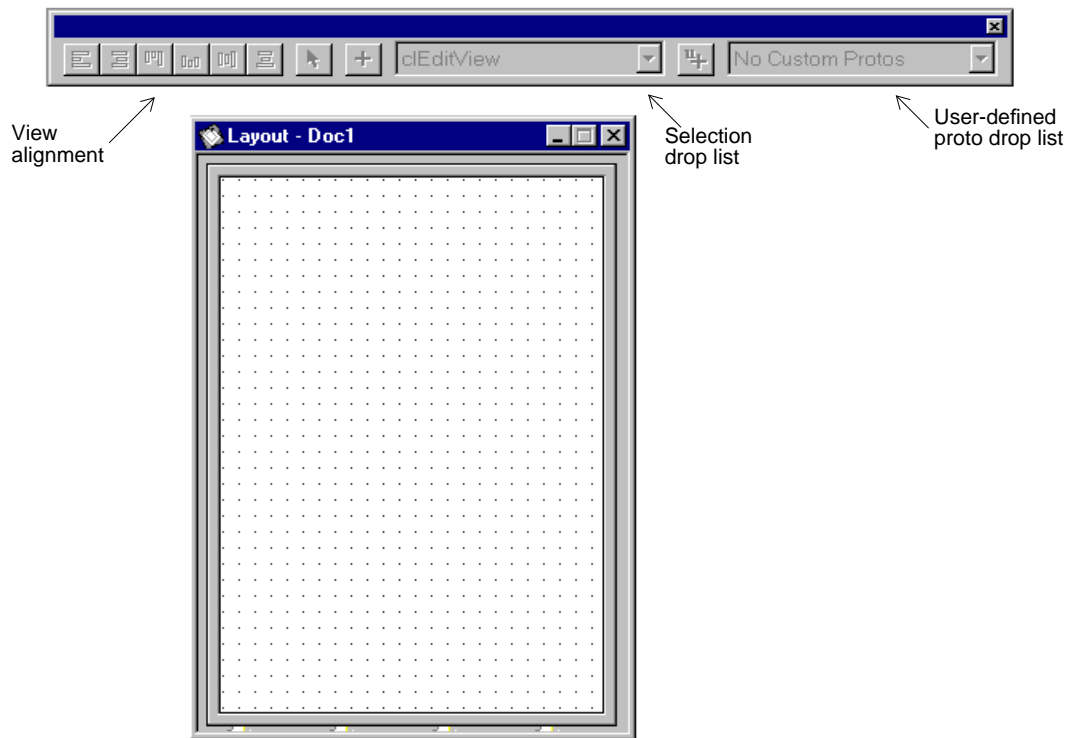
You typically start with the **application base view**, the screen image that appears when the user starts the application. The application base view is the ancestor of all other views in an application. It's the hub of the application, both visually and structurally.

1. Choose New Layout from the File menu.

## A Quick Tour of NTK

NTK displays a layout window and a toolbar as illustrated in Figure 3-1, which contains the Selection drop list and User-defined proto drop list to assist in the creation of templates.

**Figure 3-1** Layout window and toolbar



The default layout window represents the screen of the Newton MessagePad. Applications built for the MessagePad platform execute on all Newton devices, although they do not support features exclusive to the Newton 2.0 platform.

## A Quick Tour of NTK

You can complete this tutorial using either platform. The default screen sizes are slightly different.

1. Move the mouse to the Selection drop list, and click the button.

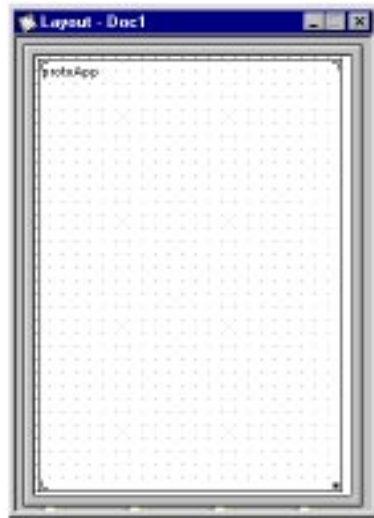
NTK displays the names of all view templates built into the Newton and available through the selected platform file.

2. Select `protoApp`.

The `protoApp` proto defines a view with a few basic application features: a title bar, a status bar, and a close box.

Most applications use either the `protoApp` proto or the `clView` view class for the application base view. For descriptions of the system-defined protos and view classes, see the *Newton Programmer's Guide*.

3. Draw the base layout view, positioning it approximately as shown here.



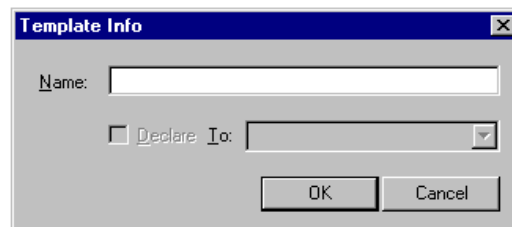
The rectangle you define here determines the size and location of the application on the Newton screen.

## A Quick Tour of NTK

NTK labels your new layout view `protoApp`, using the name of the proto from which it was constructed.

4. Choose Template Info from the Browser menu.

NTK displays the Template Info dialog box, which lets you name and declare view templates. (Declaring a view allows you to access the view symbolically from another view, as described briefly in “Naming and Declaring Views” beginning on page 5-13 and in more detail in the “Views” chapter in *Newton Programmer’s Guide*.)



5. Type the name `helloBase`, and then click OK.
6. Save the layout file by choosing Save As from the File menu, typing the name `Hello`, and then clicking Save.
7. Choose Add Hello from the Project menu.  
NTK adds the layout `Hello` to the project file.
8. Activate the project window (by clicking the title bar or choosing the project `Hello` from the Window menu) to see the list of files in the project, which at this point includes only the file `Hello`.



NTK designates the first layout file you add to an application the **main layout file**, that is, the layout file containing the application base view,

## A Quick Tour of NTK

identified in the project window by a bullet next to its name. You can change the designated main layout file through the Project menu.

9. Choose Save from the File menu to save the project file.

## Laying Out Application Elements

---

You lay out the elements of an application within the application base view. In this section of the tutorial, you add a view that accepts handwritten input.

1. Activate the layout window, now titled Hello, by clicking its title bar.
2. Using the Selection drop list, select `protoLabelInputLine`.
3. Draw out a rectangle inside the application base view, imitating the size and location shown here.





## Customizing a View Template

---

When you lay out a view, NTK creates a **view template**, a frame containing the named slots that define the view.

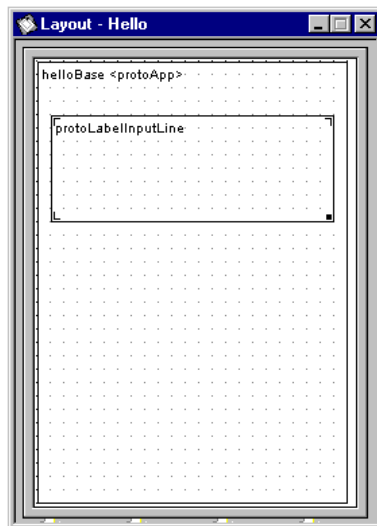
You can edit view templates with NTK's **browser**. The browser displays a list of templates and a list of slots within a selected template. When you select a slot for editing, the browser invokes a slot editor of the appropriate type.

### Editing a Slot

---

In this section, you change the application's title by editing the `title` slot in the application base view template.

1. Select the application base view by clicking within the `helloBase` view but outside the `protoLabelInputLine` view. Small selection marks appear in the corners of the selected view.

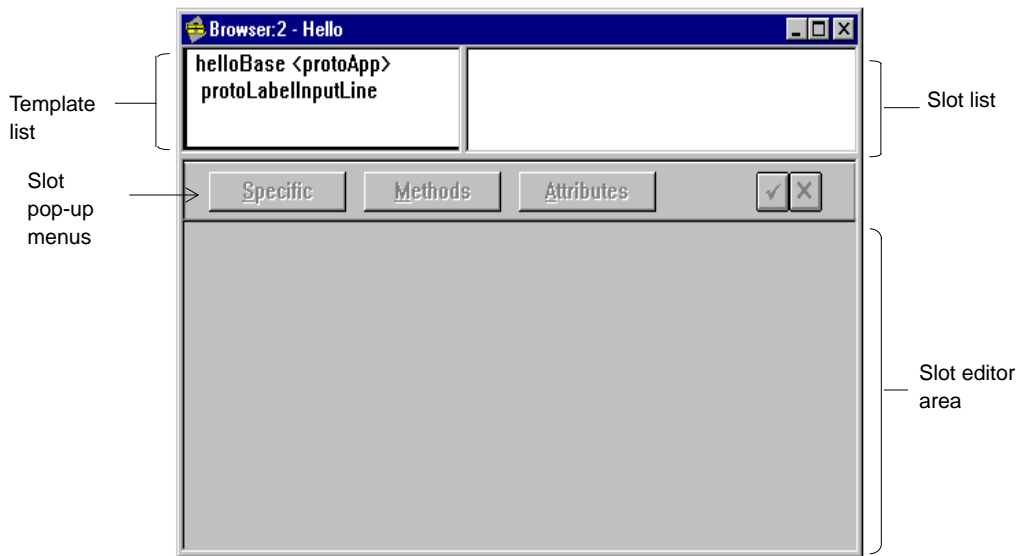


## A Quick Tour of NTK

2. Choose New Browser from the Window menu.

NTK displays a browser window for the base view, as illustrated in Figure 3-2.

**Figure 3-2** A browser window



The template list shows the templates for the selected view and all its children. The slot list shows the slots for the view that's selected in the template list. The slot pop-up menus list system-defined slots you can add to your templates. When you open a slot for editing, you work in the slot editor area.


3. Click helloBase<protoApp> to select it.

## A Quick Tour of NTK

NTK displays the slot list for that template.

```
_proto
title
viewBounds
viewFormat
```

4. Click the `title` slot to open it for editing.  
NTK displays the slot's current contents, "Application", in the editing area.
5. Change the default text to  
"Hello"
6. Apply the change by clicking the Apply check mark in the center-right of the browser window.

Apply  Revert

When you apply a change to a slot that contains code, NTK checks the syntax. It displays an alert if it finds any syntax errors, but it applies the change in any case.

You can also apply a change by pressing Ctrl-E or choosing Apply from the Browser menu.

7. Save the layout file by choosing Save from the File menu.  
Saving with either the browser window or a layout window active saves the associated layout file.

## Adding a Slot

---

You can add system-defined slots to a view through the pop-up menus labeled Specific, Methods, and Attributes, located below the slot list in a browser window.



## A Quick Tour of NTK

You can also define and add your own slots by choosing New Slot from the Browser menu, as described in “Adding Slots” beginning on page 5-18.

In this section of the tutorial, you add a label to the application’s pen-input view by adding a `label` slot to the view template.

1. In the template list in the browser window, select `protoLabelInputLine` by clicking it.  
NTK displays the slots defined for that view, `_proto` and `viewBounds`.
2. Choose `label` from the Specific pop-up menu.  
NTK invokes the text editor and displays the default label, "Label". It adds the `label` slot to the slot list.
3. Replace the default label with your own text, such as  
"Write Here"

## Building and Downloading a Package

---

At almost any point after you’ve laid out an application base view, you can build your application into a package, which you can download and run on a Newton.

1. Choose Build Package from the Project menu, or press Ctrl-1 on your keyboard.  
NTK builds the package and places it in the project directory. NTK places the package in a file called *projectname.pkg*.  
By default, NTK saves all files in a project before building. You can change this feature through the Toolkit Preferences-Packages item in the Edit menu.

## A Quick Tour of NTK

2. Choose Download Package from the Project menu.

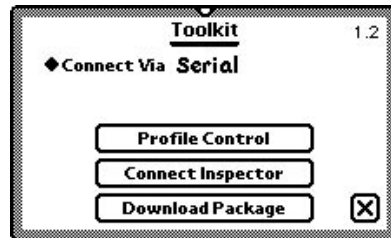
**Note**

This tutorial assumes there is no Inspector connection open but that you've set the communication settings on both the Newton and the development system and downloaded the Toolkit application, as described in "Installing the Toolkit Application on the Newton" beginning on page 1-3.

If you've made an Inspector connection, the Inspector handles the downloading from this point, and you can skip to step 5. ♦

The development system reports its communication settings and prompts you to initiate the connection on the Newton.

3. Tap the Toolkit icon in the Extras drawer on the Newton.  
The Toolkit application opens.



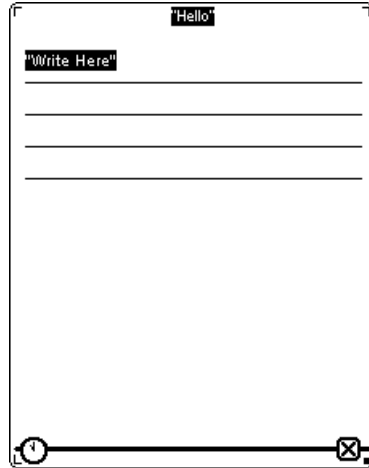
4. Tap Download Package.

The development system reports progress during the download. When downloading is complete, the application appears in the Extras drawer on the Newton.



## A Quick Tour of NTK

5. Open the application by tapping its icon.



6. Test the application by writing in the input view.

## Adding a Linked Layout

---

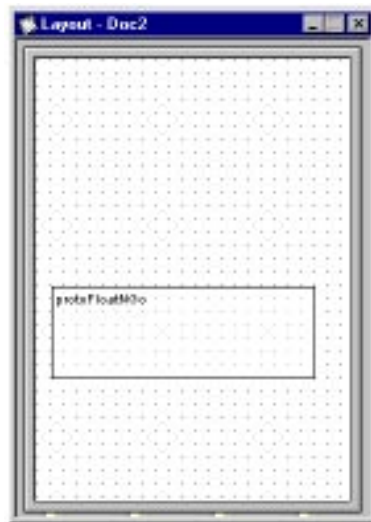
As an application grows more complex, the layout window can become cluttered. You can split your application into logical modules and keep your layout windows manageable by laying out child views in separate template files and linking them into the application through an element called a `linkedSubview`, available through the Selection drop list in the NTK toolbar.

In this tutorial, you lay out a floating window in a separate layout file and link it to the application base view. You bring it into the interface by adding to the base view a button that, when pressed, sends an `Open` message to the linked view.

## Laying Out a Linked View

---

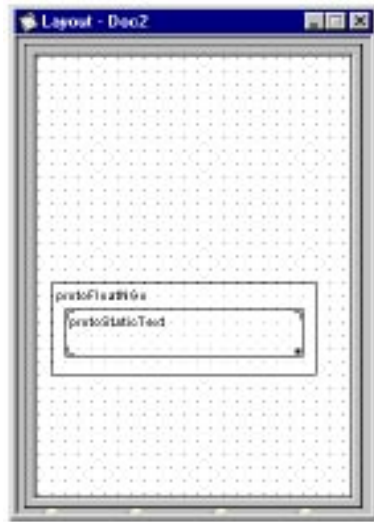
1. Choose New Layout from the File menu.
2. Using the Selection drop list, select `protoFloatNGo` from the menu.  
Like all layout files, a layout file for a linked view must have a main layout view—in this case, a view based on `protoFloatNGo`—which is the parent of all other views in the file.
3. Draw the view, positioning it approximately as shown here.



4. Choose Template Info from the Browser menu, name the view `floaterLink`, and then click OK.
5. Using the Selection drop list, select `protoStaticText`.

## A Quick Tour of NTK

6. Lay out the text view within the layout view approximately as shown here.



This rectangle defines the location of the static text message within the linked view.

7. With the `protoStaticText` view still selected, choose New Browser from the Window menu. NTK displays a new browser.



8. Click `protoStaticText` to display the slots in that template.
9. Click the slot name text to open the slot for editing.  
The browser displays the default text, "Static Text".
10. Select the default text and replace it with your own message, such as  
"Hello, world, from a linked view"
11. Close the browser window.



## A Quick Tour of NTK

NTK automatically applies pending slot changes when you close a browser window, when you save a file, or when you open a different slot for editing.

12. Save the layout file with the name `floatMsg`.
13. Choose `Add floatMsg` from the Project menu to add the layout to the project.
14. Activate the project window by clicking its title bar or by choosing `Hello` in the Window menu.
15. Select the file `floatMsg` and then choose `Process Earlier` from the Project menu or press `Ctrl-Up Arrow` to move it ahead of the main layout in the project list.
16. Save the project file.

## Linking in the Layout

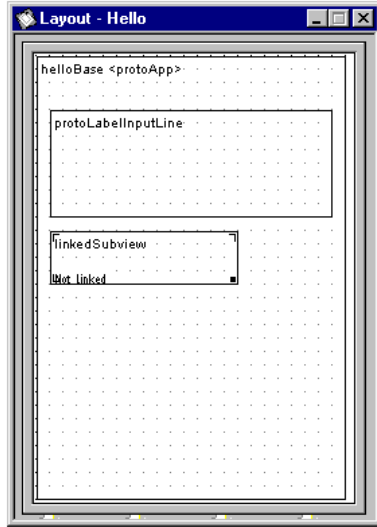
---

You link an external layout into an application by adding a special element called a linked subview to the main layout file and making a link between that element and the external file.

1. Activate the main layout window, `Hello`.
2. Select `linkedSubview` from the Selection drop list.

## A Quick Tour of NTK

3. Lay out the linked subview approximately as shown here.



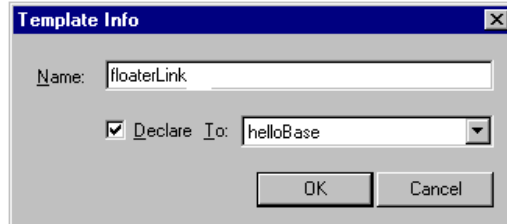
The placement of the linked subview element doesn't matter. The position of the view itself is determined by the linked template (in this case, the `floaterLink` template in the layout file `floaterMessage`).

4. With the `linkedSubview` view still selected in the layout window, choose Template Info from the Browser menu.
5. Type in the name `floaterLink`, but don't click OK yet.

This tutorial uses the same name for the linking view and for the layout view in the external layout file because the two templates share the same place in the view hierarchy. "Linking Multiple Layouts" beginning on page 5-14 explains how NTK processes linked subviews and the layout files they're linked to.

## A Quick Tour of NTK

6. Click the box labeled Declare.



Declaring the linked subview is not necessary for linking. You declare the view in this step so that the button you add in the next section can send an Open message to its sibling, the `floaterLink` view.

7. Click OK.
8. With the `floaterLink` view still selected, choose Link Layout from the File menu.

NTK displays the file-select dialog box.

9. Select the filename `floatMsg`, and then click OK.

The Hello layout window now reflects that `floaterLink` is linked to the linked view `floatMsg`.

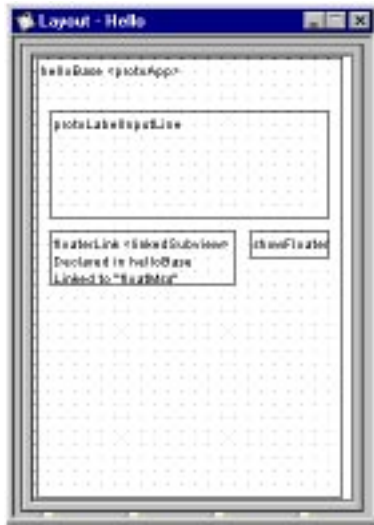
## Adding a Button That Displays the View

To incorporate the `floater` view into the application's interface, you add to the application's base view a button that sends an Open message to the linked view when it's tapped.

1. With the Hello window still active, select `protoTextButton` from the Selection drop list.

## A Quick Tour of NTK

2. Draw the button, positioning it approximately as shown here.



The rectangle you draw in this step determines the size and position of the button on the Newton screen. All descendants of the application base view must be contained entirely within the application base view; any portions that fall outside aren't visible on the Newton.

3. Use Template Info in the Browser menu to name the new view `showFloaterButton`.
4. Activate the Hello browser window.
5. Select `showFloaterButton<protoTextButton>` from the view list, and then click the `buttonClickScript` slot to open it for editing. NTK invokes the script slot editor and displays a skeletal function statement.
6. Insert an instruction to send an Open message to the `floaterLink` view:

## A Quick Tour of NTK

```
func ( )
begin
    floaterLink:Open ( ) ;
end
```

7. Click the text slot to open it for editing.  
This slot specifies the text on the button itself.
8. Change the default button text to  
"Show Linked View"
9. Save the file.
10. If you want to test the linked view, build and download the application as described in "Using the Inspector" beginning on page 3-30.

## Defining Your Own Proto

---

This section of the tutorial creates a **user proto**—a proto defined by you, not built into the Newton—that passes data among views.

If you needed this template in only one place, you'd likely lay it out as a standard layout file. Defining a layout as a proto, however, opens up two possibilities:

- You can use the same template in different views.
- Your application can use the proto to create views as needed at run time.

### Laying Out a Proto and Adding It to the Toolbar

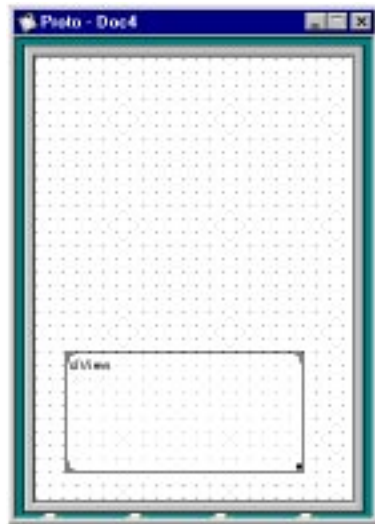
---

1. Choose New Proto Template from the File menu to open a proto layout window.

You set up a proto template the same way you set up any other layout file: you establish the layout base view and place other elements within it.

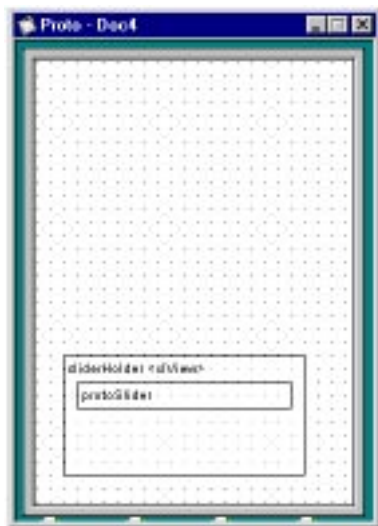
A Quick Tour of NTK

2. Select `clView` from the Selection drop list.  
The `clView` view class is the most basic container view.
3. Draw the layout base view approximately as shown here.



## A Quick Tour of NTK

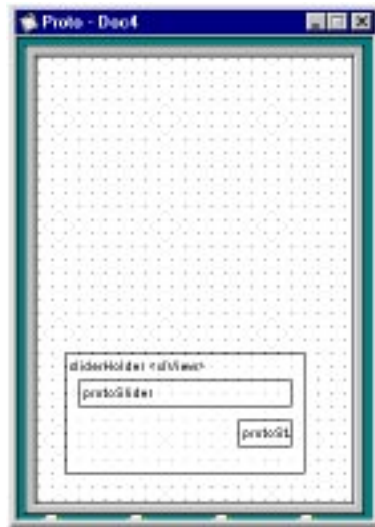
4. Use Template Info to name the template `sliderHolder`.
5. Choose `protoSlider` from the Selection drop list.
6. Draw a wide, shallow rectangle within container view.



7. Select `protoStaticText` from the list.

## A Quick Tour of NTK

8. Draw a static text view to hold the value of the slider.

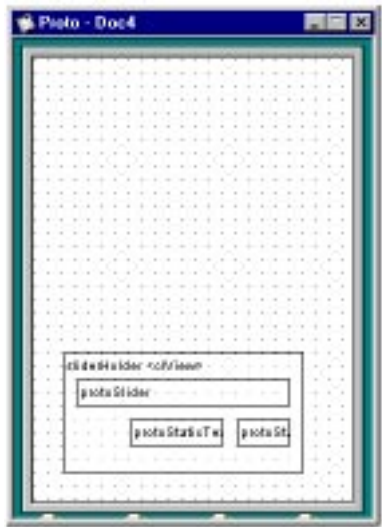


9. Use Template Info to name the view `outputView` and to declare `outputView` to `sliderHolder`.
10. Select `protoStaticText` again from the list.



A Quick Tour of NTK

11. Draw a view to hold a label.



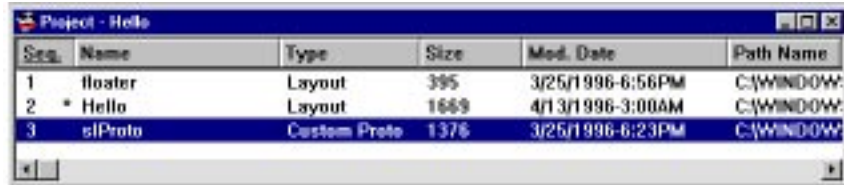
12. Save the layout file as slProto, and then choose Add slProto from the Project menu to add it to the project file.
13. Activate the project window, Hello.
14. Select slProto in the project window.
15. Choose Process Earlier from the Project menu, or press the Ctrl key and Up arrow key once to move slProto ahead of Hello in the project list.

A screenshot of a window titled "Project - Hello". The window displays a table with the following data:

Seq.	Name	Type	Size	Mod. Date	Path Name
1	flouter	Layout	395	3/25/1996-6:56PM	C:\WINDOW
2	* Hello	Layout	1669	4/13/1996-3:00AM	C:\WINDOW
3	slProto	Custom Proto	1376	3/25/1996-6:23PM	C:\WINDOW

## A Quick Tour of NTK

16. Tap the heading Seq. in the project window to display the files by their order in the build sequence.



Seq.	Name	Type	Size	Mod. Date	Path Name
1	floater	Layout	395	3/25/1996-6:56PM	C:\WINDOWS\
2	* Hello	Layout	1669	4/13/1996-3:00AM	C:\WINDOWS\
3	slProto	Custom Proto	1376	3/25/1996-6:23PM	C:\WINDOWS\

17. Save the project file.
18. In the project window, double-click the filename slProto to open a browser window, which lists the templates you've laid out.

```
cView: sliderHolder
protoSlider
protoStaticText: outputView
protoStaticText
```

19. Select the unnamed static text view protoStaticText.  
The slot list displays the slots in that template.

20. Select the text slot.

21. Change the slot contents to  
"slider value"

22. In the template list, select the static text view  
outputView<protoStaticText>.

23. In the slot list, select the text slot.

24. Change the value to  
"50"

By default, the slider begins in the middle of a 0–100 scale.

25. In the template list, select the slider view, protoSlider.

26. In the slot list, select the changedSlider slot.

27. Insert the SetValue function so that the method reads:

## A Quick Tour of NTK

```
func( )
begin
    SetValue(outputView, 'text', NumberStr(viewValue));
end
```

This line sets the value of the text slot in the outputView view to the value of the slider.

28. Save the file slProto.

## Using Your Proto

---

Once you've saved your proto and added it to the project, NTK gives you access to it through the User-defined proto drop list on the toolbar.

1. Activate the Hello layout window.
2. Click the User button to activate the User proto drop list.

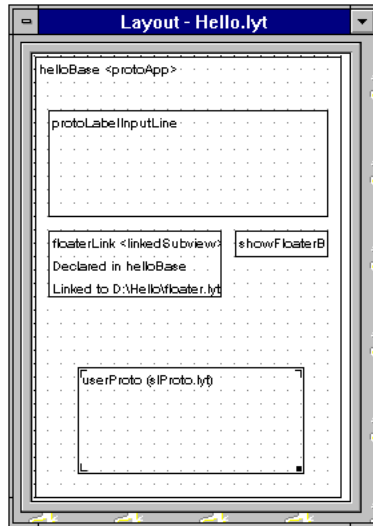


The proto `slProto`, the only item in the menu, is now selected.

3. Draw a view in the lower part of the application base view.  
Regardless of where you try to draw the new view, NTK places it where you placed the layout view for the proto `slProto`, because the view inherits its location from its proto. As with all views, you can override the

## A Quick Tour of NTK

placement by adjusting the `viewBounds` slot when the view is instantiated.



4. Build and download the package, as described in “Using the Inspector” beginning on page 3-30.

5. Open the application and test the linked template and the slider.

This section completes the laying out and coding of the tutorial application. In the rest of this chapter, you use this application to explore NTK’s debugging support.

## Using the Inspector

The Inspector is a debugging window that lets you browse the Newton object storage system and execute NewtonScript code on the Newton device.

The debugging functions used in this tutorial are documented in Chapter 6, “Debugging.”

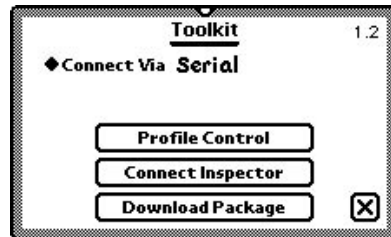
## Connecting the Inspector

---

1. On the development system, choose Connect Inspector from the Window menu.

The development system reports its communication settings and prompts you to initiate the connection on the Newton device.

2. Verify that the message describes your configuration.
3. On the Newton, tap the Toolkit icon in the Extras drawer.  
The Toolkit application opens.



4. Verify that the connection type matches your configuration.
5. Tap Connect Inspector.

The Toolkit application reports that it's opening the Inspector.

When the Inspector connection is established, the Inspector window opens on the development system.

## Executing Commands

---

Code you enter in the Inspector window is compiled on the development system and executed on the Newton device.

The Inspector compiles and executes your keystrokes only when you explicitly request it by selecting and entering text. If no text is selected when you press Enter in the numeric keypad or Ctrl-Enter on your keyboard, the Inspector processes the current line.

## A Quick Tour of NTK

1. In the Inspector window, type these characters—remember not to press Enter on the numeric keypad.

```
1 / 3 ;
```

2. With the cursor on the same line as the text, press Enter on the numeric keypad.

The Inspector displays the result of the statement in two ways: a transient reference (preceded by a pound sign) and a textual representation.

```
#441A4C1      0.333333
```

You can enter and execute any valid NewtonScript code in an Inspector window. The Newton always prints to the screen the value of the last statement evaluated.

3. Type:

```
GetRoot ( ) : SysBeep ( ) ;
```

The `GetRoot` function returns the Newton's root view. This line sends the `SysBeep` message to the root view.

4. Press Enter on the numeric keypad.

The Newton sounds the system beep, and the Inspector window displays the result of the statement.

```
#1A      TRUE
```

5. Place the two statements together on two lines:

```
GetRoot ( ) : SysBeep ( ) ;
1 / 3 ;
```

6. Select both lines and press Enter on the numeric keypad.

The Newton executes both lines but displays the result of only the last statement evaluated.

## A Quick Tour of NTK

```
#44126F1 0.333333
```

## Looking at a Frame and a View

---

This section of the tutorial looks at the Hello application developed earlier in this chapter. It assumes you have built and downloaded the complete application.

1. Open the Hello application on the Newton by tapping its icon.
2. Open the floating window by tapping the Show Linked View button.
3. Enter in the Inspector window:

```
debug("floaterLink");
```

The Inspector displays the view frame for the view instantiated from the floaterLink template.

```
#440C359 {_Parent: {_Parent: {#4407939},
                        _proto: {#600044BD},
                        viewCObject: 0x1108C45,
                        floaterlink: <2>,
                        viewBounds: {#4414E61},
                        viewclipper: 17861715,
                        base: <1>,
                        viewFlags: 5},
          _proto: {viewBounds: {#600047BD},
                  stepChildren: [#600047FD],
                  _proto: {#2D3},
                  debug: "floaterLink",
                  preAllocatedContext: floaterlink},
          viewCObject: 0x1108E20,
          base: <1>,
          viewFlags: 65}
```

## A Quick Tour of NTK

You can specify how many layers of child views and how many slots within a layer are displayed by setting the `printDepth` and `printLength` parameters, described in “Debugging Variables” beginning on page 6-21.

4. On the Newton device, close the floater window by tapping the close box.
5. Put the insertion point anywhere in the line `debug( "floaterLink" );` and then press Enter on the numeric keypad.

The Inspector responds `NIL`, because the view is not instantiated.

6. Enter in the Inspector window:

```
dv( debug( "helloBase" ) );
```

The Inspector displays the named view and its children.

```
helloBase      #44100D9 [ 10,  4,230,320] 10000005 vVisible vApplication vHasChildrenHint
|180244        #4416681 [103,  2,137, 18] 40000003 vVisible vReadOnly
|362116        #4416751 [ 10,302,230,320] 50000001 vVisible vHasChildrenHint
||291204       #44167B9 [ 14,302, 31,319] 60000201 vVisible vClickable vHasIdlerHint
||46154304     #4416841 [211,304,224,317] 40000203 vVisible vReadOnly vClickable
|93132404     #44168B1 [ 18, 52,226,140] 50000201 vVisible vClickable vHasChildrenHint
||2915956     #44168C9 [ 80, 52,226,140] 40003A01 vVisible vClickable vGesturesAllowed vCharsAllowed
||5793920     #4416919 [ 18, 54, 80, 67] 40000203 vVisible vReadOnly vClickable
|showFloaterB #4416BE9 [132,150,224,170] 40000203 vVisible vReadOnly vClickable
|sliderHolder #4416C01 [ 34,244,210,316] 50000001 vVisible vHasChildrenHint
||360448      #4416C69 [ 50,260,194,276] 40000201 vVisible vClickable
||outputView  #4416C19 [170,284,194,300] 40000003 vVisible vReadOnly
||5793920     #4416CD1 [ 98,284,162,300] 40000003 vVisible vReadOnly
|floaterLink  #4416649 [ 30,162,210,222] 10000041 vVisible vFloating vHasChildrenHint
||1451352     #4417001 [196,208,209,221] 40000203 vVisible vReadOnly vClickable
||5793920     #4417019 [ 38,178,214,218] 40000003 vVisible vReadOnly
#2           NIL
```

The numbers in the first column represent entries in a hash table used in most Newton ROMs. To display the proto and view class names instead, you can download to the Newton the file `DebugHashToName.pkg`, which is distributed with NTK.

## Making a Change in a Running Application

In this section of the tutorial, you change the button text on the Newton screen by changing the value of the `text` slot in its view frame.

1. Type the following in the Inspector window



## A Quick Tour of NTK

```
SetValue(Debug("showFloaterButton"), 'text, "Tap Here");
```

The button text changes to Tap Here.

2. Close the Hello application by tapping its close box, and then open it again by tapping the icon.

The text reverts to Show Linked View, because the change affected only the view frame that existed while the application was running, not the view template from which the view was instantiated.

## CHAPTER 3

### A Quick Tour of NTK

# Managing and Building a Project

---

You manage an application under development as an NTK **project**, that is, the collected files and specifications NTK needs to build a data package that can be installed and executed on the Newton device.

This chapter describes how you use NTK to

- set up a project and organize the files in it
- establish settings and preferences
- build a project

## Setting Up a Project

---

You manage an NTK project through the **project file**, which contains a list of the files to be processed during the project build. To start a project, you create a project file by choosing New Project from the Project menu.

## Managing and Building a Project

You can add to a project

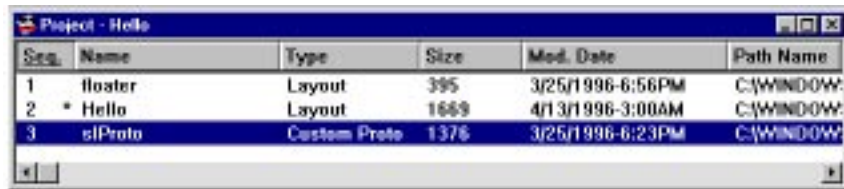
- **layout files**, which contain templates for views you've laid out with NTK's graphical editor
- **text files**, which contain optional installation and removal scripts and other NewtonScript code outside the scope of the view templates
- **bitmap** and **sound files**, which contain resources used during the build
- **package files**, which contain software ready to be installed on the Newton
- **object stream files**, which contain NewtonScript frames encoded in Newton Streamed Object Format

## Project File

---

The project file contains a collection of project settings and a list of the files to be processed during the build. When the project file is open, NTK displays the **project window**, illustrated in Figure 4-1.

**Figure 4-1** The project window



Seq.	Name	Type	Size	Mod. Date	Path Name
1	floater	Layout	395	3/25/1996-6:56PM	C:\WINDOW
2	* Hello	Layout	1669	4/13/1996-3:00AM	C:\WINDOW
3	slProto	Custom Proto	1376	3/25/1996-6:23PM	C:\WINDOW

You change project settings through the Settings command in the Project menu, and you add files with the Add This File and Add File commands.

The file marked with an asterisk—Hello in Figure 4-1—is the **main layout file**, that is, the file that contains the view at the top of an application's view hierarchy. NTK marks the first layout file you add to an application project as

## Managing and Building a Project

the main layout file; you can change the designation by selecting another file and choosing Mark As Main Layout from the Project menu.

During the build, NTK processes files in the order of their sequence numbers, shown in the first column. To rearrange files, select one file at a time and choose Process Earlier or Process Later from the Project menu. To change the sequence from the keyboard, press Ctrl-Up Arrow to move a file earlier in the build or Ctrl-Down Arrow to move it later. “Build Sequences” beginning on page 4-52 summarizes the constraints on the ordering of files.

The size column shows the space the file occupies on the development system.

You can change the order in which the files are displayed in the project window by clicking the column headings. To display the files alphabetically by name, for example, click Name. The heading that dictates the order—the Seq. heading in Figure 4-1—is underlined.

## Layout Files

---

Layout files contain the templates you’ve laid out and programmed using the graphical editor and browser, which are described in Chapter 5, “Laying Out and Editing Views.”

You create different kinds of layout files to hold

- the views and other templates your application uses on the Newton device
- view templates you’ve defined yourself—known as **user protos**—which can be available both during the build and at run time.

NTK processes the files one at a time, in the sequence you specify through the project window. Layouts that are used by other templates must be processed before the layout files that reference them. User protos, for example, must be processed before the layout files that use them. The last layout file in a build is likely to be the main layout file or a custom view template used only at run time.

“Processing a Template” beginning on page 4-53 describes how NTK processes layout files.

## Managing and Building a Project

## Text Files

---

You use text files to

- supply code to be executed when a package is installed or removed by the Newton system software
- define constants and functions you want available later in the build
- incorporate any other valid NewtonScript code outside the scope of the layout templates

If you are using the Book Maker application you can also create text files that hold text and formatting instructions for building on-line books.

“Text Files” beginning on page 4-31 contains more information on what you can put in text files and how NTK processes them.

## Bitmap and Sound Files

---

You can use NTK to incorporate bitmap and sound files into Newton software.

NTK itself uses 'BMP' files. You supply your application's icon as a 'BMP' file, and you can place 'BMP' files in picture slots through the picture slot editor described in “Editing Slots” beginning on page 5-20.

You can draw your pictures in any graphics program and then paste them as 'BMP' files that you include in your project. Additionally, you can create sound files in any sound program and save them as 'WAV' files that can be added to your project.

NTK also includes a set of compile-time functions that retrieve and manipulate 'BMP', 'WAV', and other resources. Appendix C, “Custom Bitmaps and Sounds,” describes resources and the functions that handle them. You add the resource files at the beginning of the project, and you place the code that handles them in a text file.

## Managing and Building a Project

## Package Files

---

When you build an application, NTK produces a package file—that is, a file containing software to be installed on a Newton device.

A package file consists of a header containing package information and one or more **parts** containing code and data. A part is a unit of software recognizable by the Newton, such as an application, a book, or a data store. When a package is installed on the Newton, the Newton system software automatically opens the package and dispatches the parts to the appropriate handlers.

Newton applications are stored in parts of type `form`; books are in parts of type `book`. NTK also supports a number of other part types, summarized in “Output Settings” beginning on page 4-16 and described more fully in “Output Options” beginning on page 4-47.

When NTK processes a project into a package, it produces one new part, of the specified type. You can place additional parts in the same package by putting the package files that contain them into the project. NTK places the parts in the package as it encounters them during the build: It places parts from package files that appear in the file list before the layout and text files before the new part in the final package; it places parts from package files that appear after the layout and text files after the new part. The final package has the attributes established for the current build—the new package—through the Package tab of the Settings dialog box. NTK ignores the attributes of any other package files in the project.

The order of the parts in the package determines the order in which the parts are installed and removed by the Newton system software.

## Object Stream Files

---

You can use NTK to build object stream files—that is, files encoded in Newton Streamed Object Format (NSOF). You can then incorporate these stream files into your application by adding them to the project.

## Managing and Building a Project

You can use stream files to shorten the build time by preprocessing large data structures or other static input. “Stream Files” beginning on page 4-50 explains how you can use stream files.

## Establishing Settings and Preferences

---

You use the Settings item in the Project menu to establish build specifications:

- Application Settings determine the application’s name, symbol and icon.
- Project Settings establish project-wide choices, such as the target platform.
- Package Settings determine the features of the package that’s output from a build.
- Output Settings determine what kind of part is being built, and, if the part is an application or a book, ie, the part characteristics.

You use the Toolkit Preferences items in the Edit menu—App, Layout, Browsers, Text Views, Packages, and Heaps tabs—to configure NTK for your hardware setup and working style.

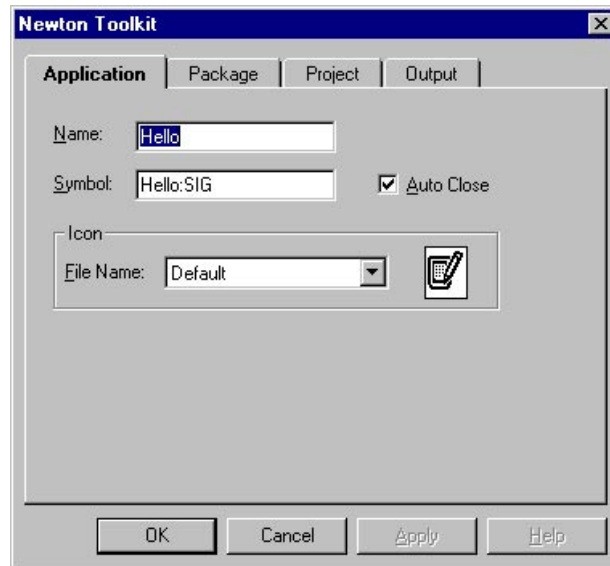
### Application Settings

---

You set the application’s name and symbol through the Application tab in the Settings dialog box, illustrated in Figure 4-2. This is the default dialog box for Settings.



## Managing and Building a Project

**Figure 4-2** Settings-Application

**Name** The text that appears beneath the application's icon in the Newton Extras drawer.

**Symbol** The application's unique symbol, the alphanumeric string by which the application identifies itself to the Newton root view.

At the beginning of the build, NTK defines a constant with the name `kAppSymbol` and sets it to the symbol you specify here.

At the end of the build, if you've not created a slot with the name `appSymbol` in the application base view, NTK

## Managing and Building a Project

creates one and places in it the symbol you specify here. If the slot exists already, NTK doesn't overwrite it.

Apple recommends you build your application symbol from the application name and your company's registered signature, using this convention:

*name:signature*

A developer with the signature SURF, for example, might identify a checkbook application with the symbol `checkb: SURF`.

To ensure uniqueness across third-party products, PIE Developer Technical Support maintains a registry of developer signatures. To register your signature, contact the registry at the addresses listed at the front of this book.

A book does not use a symbol the same way an application does; therefore, this field does not apply to books.

**Auto Close** Identifies this as an Auto Close application—that is, one that closes when another Auto Close application opens. This option is supported only on the Newton MessagePad platform.

This characteristic does not apply to books.

**Icon** The bitmap file containing the application's icon bitmap. The drop list displays all bitmap files in the project.

**File Name** A named 'BMP' file—in the specified file—that contains the icon that represents the application in the Extras drawer.

NTK can use only a 'BMP' file for the icon. You can supply a mask in a companion 'BMP' file with the same name followed by an exclamation point. If an icon is named wave, for example, NTK looks for a mask with the name wave!. (A mask is a parallel bitmap used to display the icon when it's selected. If you don't supply your own mask, NTK creates one.) Instructions for

## Managing and Building a Project

creating 'BMP' files appear in Appendix C, "Custom Bitmaps and Sounds."

The selected icon appears next to the drop lists—the default icon appears only if there's at least one resource file in the project. The standard size for icons is 29 pixels high by 31 pixels wide.

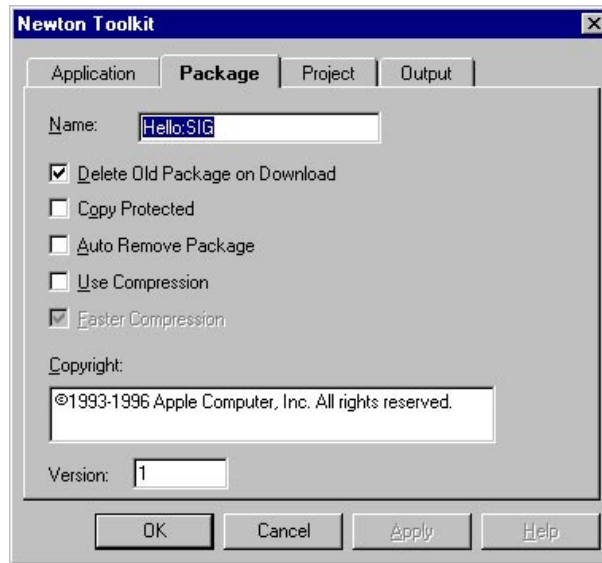
## Package Settings

---

You can use NTK to build a package file—that is, a file containing software ready to install on a Newton device—by choosing one of the package part types in the Output tab of the Settings dialog box, described in "Output Settings" beginning on page 4-16.

You specify the package name and other package characteristics through the settings that appear when you click the tab labeled Package.

## Managing and Building a Project

**Figure 4-3** Settings-Package**Name**

The package name—that is, the name of the package as it will be installed on the Newton device. Each package on a Newton must have a unique name.

Apple recommends you build your package names and application symbols from the application name and your company's registered signature, as described in the documentation of the symbol on page 4-10.

**Delete Old Package on Download**

Invokes automatic package removal when you try to download a package with the same name as a package already in place on the Newton device. If this option is in effect, NTK removes the old package and then downloads the newly built package. If this option is not in effect, you must remove an old package from the

## Managing and Building a Project

	<p>Newton before downloading a new package with the same name.</p>
Copy Protected	<p>Sets a field in the package header that marks the package as copy-protected.</p> <p>This field is a convention recognized by software that copies packages; it is not an absolute lock against copying.</p> <p>A copy-protected package can be backed up and synchronized to the desktop—users can copy the package using selective restore. The Newton ROM, however, refuses to beam or email a copy-protected package.</p>
Auto Remove Package	<p>Specifies a package whose parts are removed immediately after they're installed. The section "Parts in Auto-Remove Packages" on page 4-50 describes the impact of this option.</p>
Use Compression	<p>Specifies that the package be saved in compressed format, which takes up less space on the Newton . Software stored without compression runs faster and uses less battery power.</p> <p>This setting has no effect on the size of the package file on the development system; the code is compressed on the Newton device after downloading.</p>
Faster Compression	<p>Specifies that the package use the Newton 2.0 compression strategy, which takes up 10–15% more space on the Newton device but which decompresses significantly faster.</p> <p>This option is available only if Newton 2.0 Only is enabled in Project Settings. Applications compiled with</p>

## Managing and Building a Project

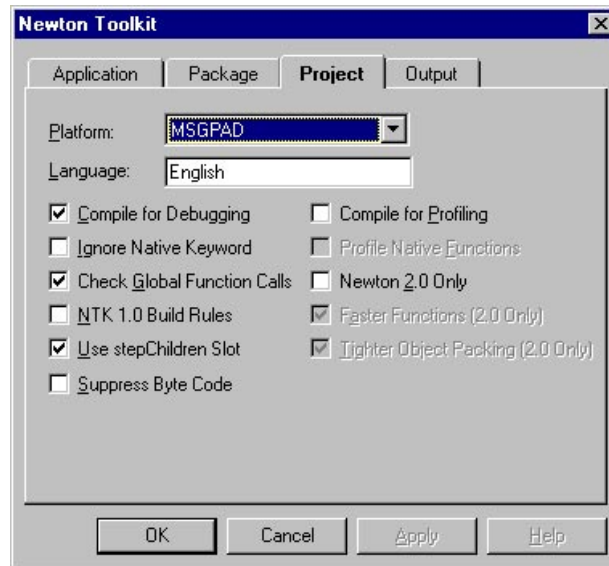
	Faster Compression enabled are incompatible with earlier Newton models.
Copyright	The copyright statement to be embedded in the package header. This text is not displayed on the Newton device.
Version	The version number to be placed in the package. In case of conflict between packages with the same name, the version number allows the Newton system software to identify the newer and older versions. This number must be an integer in the range 0 to 9999.

## Project Settings

---

You set project-wide specifications through the Project tab in the Settings dialog box illustrated in Figure 4-4.

## Managing and Building a Project

**Figure 4-4** Settings-Project

- Platform** The model of Newton on which the software will run. The drop list displays all the platform files stored in a folder with the name Platforms in the same directory as the NTK application.
- Language** The language code for use by the LocObj function. The section “Establishing a Local Language” on page 4-46 explains how NTK uses the language string. The LocObj function is described in the localization chapter in the book *Newton Programmer’s Guide*.
- Compile for Debugging** Specifies a build with embedded debugging support. When this option is enabled, the compiler adds a slot named debug to each view that you name through Template Info in the Browser menu. The value of the debug slot is the view’s name. If you create your own

## Managing and Building a Project

debug slot for a view, NTK does not override that definition.

You can test to see whether this option is enabled by testing the `kDebugEnabled` constant, which is `true` when Compile for Debugging is checked.

For more information about the Compile for Debugging option, see “Embedding Debugging Information” on page 4-44.

## Ignore Native Keyword

Suppresses the native compiler, which compiles functions defined with the `func native` syntax into ARM machine code. For more information about compiling into native code, see “Compiling Functions for Speed” beginning on page 8-10.

You can test to see whether this option is enabled by testing the `kIgnoreNativeKeyword` constant, which is `true` when Ignore Native Keyword is checked.

## Check Global Function Calls

Leaves the compiler’s global-function checking intact. When NTK compiles a global function, it checks the call against its own table of global functions and reports discrepancies in the Inspector window. This check is for your information only; the outcome has no effect on the build.

This option lets you suppress messages regarding global functions you’ve defined yourself.

## NTK 1.0 Build Rules

Invokes these build conventions from earlier releases of NTK:

- As the last step in the build, NTK processes unused user protos and places them in a slot in the base view. The name of the slot is the name of the proto layout file, with the prefix `pt_`. A proto saved in a file with the



## Managing and Building a Project

name `HandyView`, for example, would be placed in a slot with the name `pt_HandyView`.

- NTK does not define the constants `kAppName`, `kAppString`, `kAppSymbol`, and `kPackageName`, described in Table 4-1 on page 4-34.

Use `stepChildren` Slot

Instructs the compiler to place the views created by a view's children in a slot named `stepChildren`. If this checkbox is not checked, the compiler uses the name `viewChildren` instead.

Disabling this option is never appropriate when you're building software. As explained in the "Views" chapter of *Newton Programmer's Guide*, you must place child views in the `stepChildren` slot.

## Suppress Byte Code

Instructs the compiler to omit from the output the byte code version of a function compiled into native code. This option is not meaningful if Ignore Native Keyword is selected.

For more information about compiling into native code, see "Compiling Functions for Speed" beginning on page 8-10.

## Compile for Profiling

Turns profiling on. While this option is enabled, NTK includes profiling support in any package it builds.

For a description of the profiler, see "Measuring Performance" beginning on page 8-1.

You can test to see whether this option is enabled by testing the `kProfilingOn` constant, which is `true` when Compile for Profiling is checked.

## Profile Native Functions

Specifies individual profiling of functions compiled into native code.

As explained in "Profiling Native Functions" beginning on page 8-19, the distortion added by the profiling code itself is especially noticeable in the execution of native

## Managing and Building a Project

functions called by other native functions. If this option is not checked, the profiler reports only those calls to native functions that are made from interpreted functions; the time reported includes time spent in any other native functions called from that function. If this option is checked, the profiler tracks and reports all native function calls.

## For Newton 2.0 Only

Makes available a number of options that are compatible only with the Newton 2.0 platform. Enabling this option alone has no effect on the build. Disabling this option invalidates the settings of all options that are compatible only with the Newton 2.0 platform.

## Faster Functions (2.0 only)

Enables 2.0-style functions, which execute faster on the Newton 2.0 platform. Functions compiled with this option enabled are incompatible with earlier Newton models.

## Tighter Object Packing (2.0 only)

Align objects on four-byte boundaries instead of eight-byte boundaries. The application takes up 3–5% less space on the Newton with this option in effect.

Applications compiled with this option enabled are incompatible with earlier Newton models.

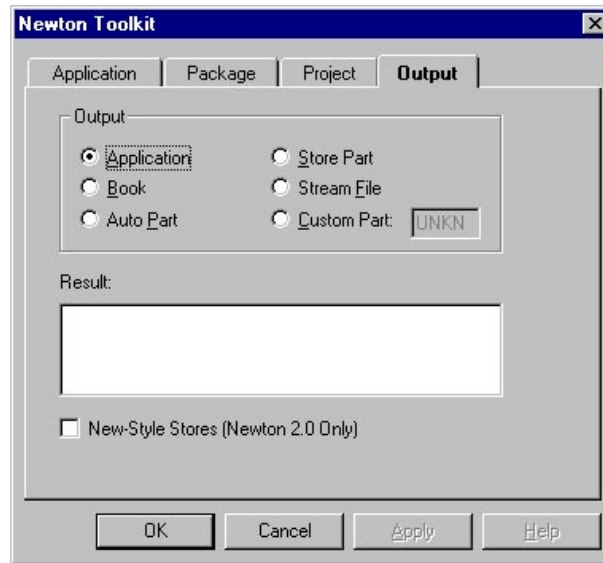
## Output Settings

---

In a single build, you can create either a package file or an object stream file. A package file holds software to be installed on the Newton device. An object stream file holds a hierarchy of NewtonScript frames encoded in Newton Streamed Object Format.

A package file consists of a header containing package information and one or more parts containing code and data. Each build produces one part, with the characteristics you set through the Output tab of the Settings dialog box, illustrated in Figure 4-5.

## Managing and Building a Project

**Figure 4-5** Settings-Output

To display the output settings, click the icon labeled Output in the dialog box that appears when you choose Settings from the Project menu.

## Output

The Output selection determines whether a build produces a package file or an object stream file and—if a package file—what part type it produces.

## Managing and Building a Project

You can use NTK to build any of four standard part types, an object stream file, or an arbitrary part type you specify.

Application	A part of type <code>form</code> , which is usually an application that's installed in the Newton Extras drawer.
Book	A part of type <code>book</code> , which contains a book file to be processed by the Newton book reader. Parts of type <code>book</code> are installed in the Extras drawer on the Newton.
Auto Part	A part of type <code>auto</code> , which contains only an installation script and a remove script. You use <code>auto</code> parts to hold software that is not associated with a visible element in the Newton Extras drawer. When the package is downloaded, it is dispatched to the package handler, but nothing is placed in the Extras drawer.
Store Part	<p>A part of type <code>soup</code>, which contains a store. If Store Part is selected here, NTK makes available a global variable named <code>theStore</code>, which contains a store. It generates a part of type <code>soup</code> that contains all data written to <code>theStore</code> during the build.</p> <p>If you choose Store Part and For Newton 2.0 Only is enabled in Project Settings, a checkbox labeled New-Style Stores appears in the dialog box. If your software is intended to run exclusively on the Newton 2.0 platform, check the New-Style Stores option.</p>
Stream File	<p>A file in Newton Streamed Object Format.</p> <p>If you choose Stream File, a field labeled Result appears in the dialog box. You must enter in the Result field an expression that evaluates to the top-level frame of the output.</p>
Custom Part	<p>A part of the type you specify here with a four-character code.</p> <p>If you choose Custom Part, a field labeled Result appears in the dialog box. You must enter in the Result field an expression that evaluates to the top-level frame of the output.</p>

## Managing and Building a Project

---

## Result

**Result**                      The Result field appears only if you choose Object Stream or Custom Part among the Output options. You must enter in the Result field an expression—typically a global variable—that evaluates at the top-level frame of the output.

**New-Style Stores**[Newton 2.0 only]  
Description needed.

“Output Options” beginning on page 4-47 discusses the output options in more detail.

---

## Toolkit Preferences

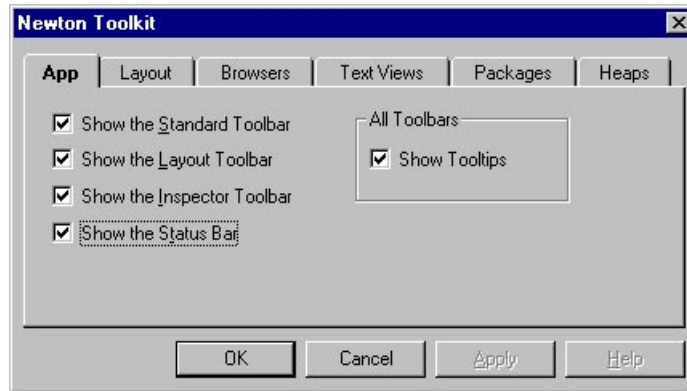
You use the Toolkit Preferences dialog box to

- set your build preferences
- identify a type and port for the connection between the development system and the Newton
- build a project

## App Preferences

---

**Figure 4-6** Toolkit Preferences-App



The App settings of the Toolkit Preferences dialog box, as shown in Figure 4-6, allow you to establish personalized views of the NTK toolbars.

### Show the Standard Toolbar

When checked, the Standard Toolbar is displayed. The Standard Toolbar displays icons for some commonly used menu items, such as creating a new layout, proto template, or text file; cut, copy, and paste; and building and downloading a package.

### Show the Layout Toolbar

The Layout Toolbar allows you to manipulate alignment, template views, and proto views.

### Show the Inspector Toolbar

This toolbar appears only when using the Inspector; the

## Managing and Building a Project

icons represent menu commands relating to the use of the Inspector.

Show Tooltips      Tooltips display the appropriate menu command for an icon. To view the menu command, simply place the cursor over the icon and an information box will appear within seconds.

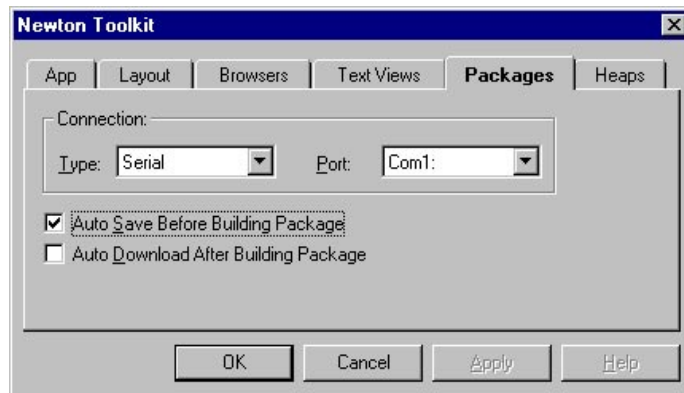
You can place the toolbars anywhere within the file window by dragging and dropping to the appropriate spot.

## Packages Preferences

---

The Packages Preferences shown in Figure 4-7 provides for communication settings between your development system and the Newton device as well as for establishing build preferences.

**Figure 4-7**      Toolkit Preferences-Packages



## Managing and Building a Project

## Connection

---

The connection settings establish the communications protocol and port for the connection between the development system and the Newton device.

Type	The communications protocol for the connection with the Newton device.
Port	The communications port for the connection with the Newton device.

## Build

---

The bottom two settings establish what NTK does automatically when building a package.

### Auto Save Before Building Package

Invokes automatic saving of all open files in the project before NTK builds a package. If any of the files have never been saved, NTK prompts you for filenames.

### Auto Download After Building Package

Invokes automatic downloading of the package to an attached Newton device after a build.



## Managing and Building a Project

Heaps Preferences

---

---

**Figure 4-8** Toolkit Preferences-Heaps

The Heaps tab of the Toolkit Preferences dialog box, as illustrated in Figure 4-8, establishes the sizes of the two NTK heaps.

**Main Heap**      The size of the main frames heap that's created when you launch NTK. The main frames heap holds your frame data while you're working in NTK.

Note that a change to this setting doesn't take effect until you quit and restart NTK.

**Build Heap**      The size of the heap that holds application frame data during the build. This heap is created each time you build. It needs to be slightly larger than the package being built.

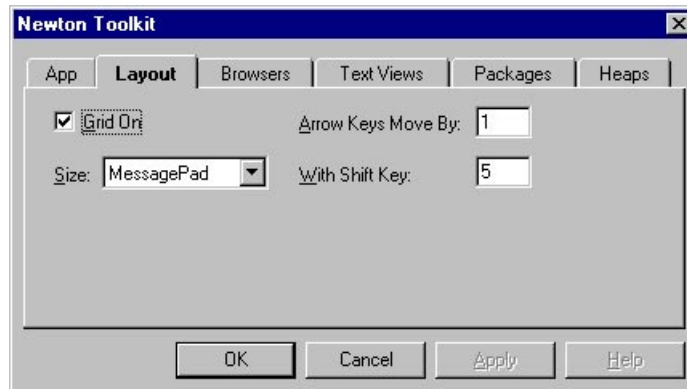
Should NTK run out of heap space during a build, it will notify you and ask for confirmation to automatically resize the build heap.

## Managing and Building a Project

## Layout Preferences

You can adjust the features of the graphical editor through the Layout preferences of the Toolkit Preferences dialog box, illustrated in Figure 4-9.

**Figure 4-9** Toolkit Preferences-Layout



Changes in your layout preferences affect only layouts that you create after making the changes.

- |         |  |
|---------|--|
| Grid On | Turns on autogrid in new layout windows. Autogrid constrains the placement and sizing of views to align with a layout grid. You can control the resolution of the grid, and you can turn the grid on or off for an individual window, through the Layout menu. |
| Size    | Establishes the initial size of layout windows. You can choose one of the existing MessagePad models. Regardless of the Screen size setting, you can change the  |

## Managing and Building a Project

size of an individual layout through the Layout Size item in the Layout menu.

## Arrow Keys Move By

Sets the number of pixels by which the selected view is moved or resized when you press an arrow key.

## With Shift Key

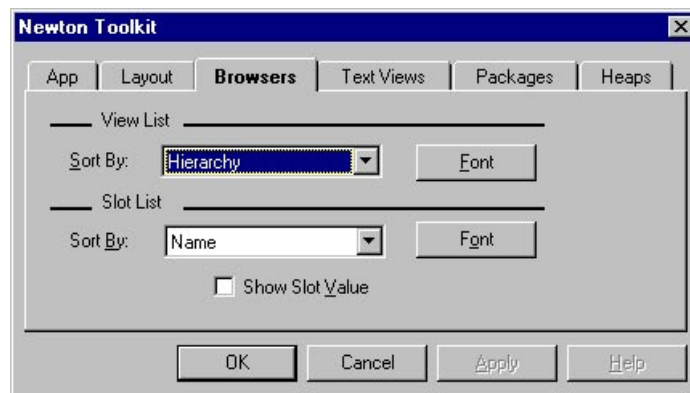
Sets the number of pixels by which the selected view is moved or resized when you press an arrow key while holding down the Shift key.

## Browser Preferences

---

You can adjust the display of templates and slots in browser windows and the characteristics of the text editor through the Browser tab of the Toolkit Preferences dialog box, illustrated in Figure 4-10.

**Figure 4-10** Toolkit Preferences-Browsers



Changes in your browser preferences affect only windows that you create after making the changes.

Managing and Building a Project

Browsers

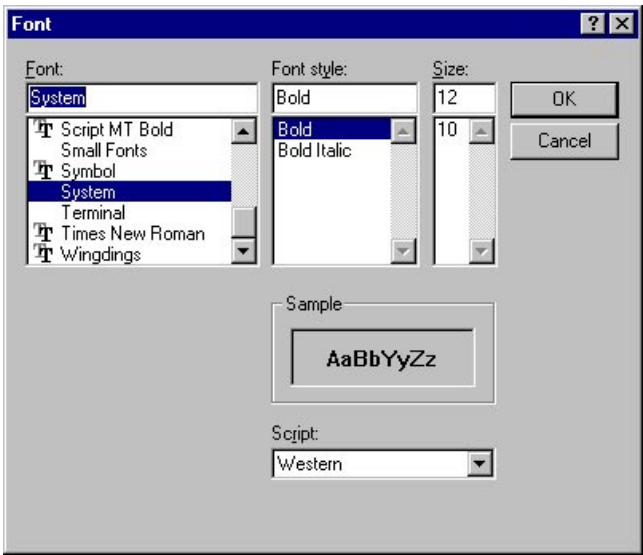
The browser settings control the ordering and font style of the view list and slot list.

View List

The view list settings control the display of the view list.

- Sort By                      Chooses a sort order for names in the view list. The Name option sorts templates alphabetically by proto name; Hierarchy sorts templates by their position in the parent / child hierarchy.
- Font                          Displays a dialog box, illustrated in Figure 4-11 through which you specify the font style in which the template names are displayed.

Figure 4-11      The Text Style dialog box



## Managing and Building a Project

**Slot List**

---

The slot list settings control the display of the slot list.

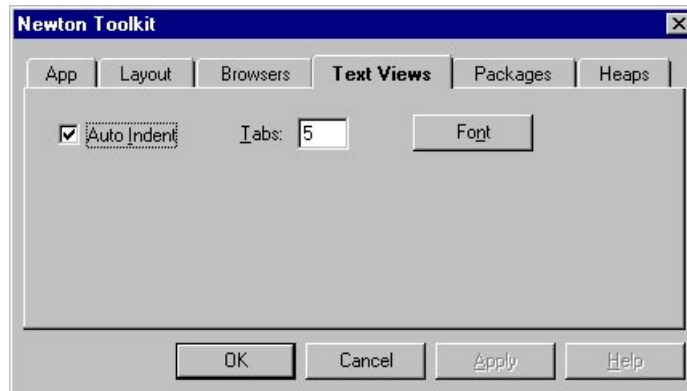
Sort By	Chooses a sort order for names in the slot list. The Name option orders slots alphabetically by name; Type orders slots alphabetically by data type.
Font	Displays a dialog box, illustrated in Figure 4-11, through which you specify the font style in which the slot names are displayed.
Show Slot Value	Invokes the display option that shows the slot value after each slot name.

**Text Views Preferences**

---

The Text Views settings control the characteristics of the text editors you use for editing slots and files.

## Managing and Building a Project

**Figure 4-12** Toolkit Preferences-Text Views

Auto Indent	Enables automatic indenting, in which the editor automatically indents a new line to the indent of the previous line.
Tabs	Sets the width of a tab, in spaces.
Font	Displays a dialog box, illustrated in Figure 4-11 on page 4-26, through which you specify the style in which text is displayed.

## Building a Project

---

NTK compiles and executes NewtonScript code, and processes templates and data files, to produce a data object that can be used by the Newton devices. The compiler compiles the text in the various source files, and the interpreter executes the resulting code at predetermined points. The code that executes during the build creates the objects that are placed in the

## Managing and Building a Project

application. Objects in memory during the build do not necessarily exist at run time.

Some of your code executes on the Newton device, some on the development system, and some in both places. The Newton and the development system run essentially the same interpreter, but the kinds of commands executed tend to be different. Some functions—such as those that handle bitmap or sounds—are available only on the development system. The *Newton Programmer's Guide*, which documents the other Newton programming functions, identifies functions that are available only at compile time. The resource-handling functions are described in this book in Appendix C, “Custom Bitmaps and Sounds,” Compile-time functions that address the mechanics of building software are described in this chapter in “Compile-Time Functions” beginning on page 4-36.

## The Build Environment

---

This section describes

- the global data file, which is compiled and executed when you open NTK
- the platform files, which contain platform data and utility functions
- the role of text files in a project
- the variables and constants NTK defines for you

## Global Data File

---

The global data file is an adjunct to NTK. You can place in it NewtonScript code that you want available from any project.

The global data file—an optional text file with the name `GlbIData.f` stored in the same directory as the NTK application—is compiled and executed once each time you open NTK. Objects you create in the global data file are available at compile time to any project you build.

NTK treats the entire global data file as if it were the body of one function with no arguments. Because NewtonScript treats variables created within a

## Managing and Building a Project

function as local, you cannot define a function in the global data file with a simple assignment statement:

```
myFunction := func(x) . . .
```

To assign a function to a variable you must either

- assign it to a slot, for example:

```
vars.myFunction := func(x) . . .
```

or

- specifically declare it as a global function:

```
global myFunction (x) . . .
```

If the compiler encounters an error in the global data file, it displays the error type, filename, and line number. When you dismiss the error dialog box, NTK quits. The prudent course is to develop code in a text file included in a project and move it to the global data file only when it's debugged. If you cannot launch NTK because of a problem in the global data file, you can bypass it by removing the file from the NTK folder.

## Platform Files

---

The platform files—stored in a directory named `Platfrms` in the same directory as NTK—contain data tailored to different Newton products. The platform files also contain a collection of Newton system definitions, a number of utility functions, and definitions for constants that reference the functions. The constant that represents a function is the function name with the prefix `k` and the suffix `Func` (that is, `kfunctionnameFunc`).

The definitions file for each platform lists the functions in that platform file.

The platform file functions are available at compile time; you can make them available at run time by incorporating them into your application in one of two ways:

- In your application base view, define an evaluate slot with the same name as the function and initialize it to the corresponding constant. For example, to use the `NewInfo` function, you create a slot named `NewInfo`



## Managing and Building a Project

and set its value to `kNewInfoFunc`. You can then call the function by sending a message to the application base view, for example:

```
:NewInfo(arg1, arg2);
```

- Call the function with the NewtonScript `call` syntax or the `Apply` function. This strategy saves space and time, because it does not require a slot in the base view and avoids inheritance lookup; it also works in code that doesn't have access to your base view, such as the remove script. Here is an example of using the `call` syntax to call a platform file function:

```
call kNewInfoFunc with (arg1, arg2);
```

This strategy does not work for functions used as methods, which must be invoked using message sending. The *Newton Programmer's Guide* identifies which functions are used as methods and which are global functions. Currently, none of the platform file functions is a method.

## Text Files

---

You use text files to

- supply code to be executed when a package is installed or removed by the Newton system software
- define constants and functions that you want available later in the build
- incorporate any other valid NewtonScript code outside the scope of the layout templates

NewtonScript code in a text file is compiled and executed when NTK processes the file. Objects you create in a text file are available throughout the rest of the build.

You can use the NewtonScript `constant` syntax to create constants with literal values. This line, for example, creates a constant named `kConst` with a value of 32:

```
constant kConst := 32;
```

When you use one of these constants as a value, NewtonScript substitutes the literal value for the constant, as described in *The NewtonScript Programming Language*.

## Managing and Building a Project

You can use the compile-time function `DefineGlobalConstant`, described in “Defining Global Constants” beginning on page 4-37, to set a global constant equal to a function.

### Install Scripts

---

An **install script** is an optional block of code that’s executed when an application is installed on the Newton device—when the card containing it is inserted, for example, or when the application’s package is downloaded.

After processing all files in the project, NTK looks for a variable with the name `InstallScript`. If it finds one, NTK uses it to build an install script, which it places in the part frame in a slot named `InstallScript`. Only some part types use install scripts. (A part is defined in “Package Files” beginning on page 4-5.)

In the case of an application, the install script is a function with one argument:

`InstallScript(partFrame)`

*partFrame*                      The part frame for the application. This frame has a slot named `theForm`, which contains a reference to your application’s base template.

The following install script, for example, registers an application with the system Find and Intelligent Assist services.

```
InstallScript := func(partFrame)
begin
    RegFindApps(kAppSymbol);
    partFrame.result := regTaskTemplate(myTemplate);
end;
```

An install script that makes changes to the system—like the example here—must be accompanied by a remove script that reverses the changes, as illustrated in the example in the next section.

In the case of an auto part, the install script is a function with two arguments:

## Managing and Building a Project

```
InstallScript (partFrame, removeFrame)
```

<i>partFrame</i>	The part frame for the auto part. This frame contains no theForm slot.
<i>removeFrame</i>	A frame that will be passed to the remove script, described in the next section. The remove frame contains a single slot, which itself contains a copy of the remove script.

An auto part must have an install script. “Auto Parts” beginning on page 4-49 describes auto parts.

### Remove Scripts

---

A **remove script** is an optional block of code that runs when your application is removed from the Newton device—when you eject the card that’s holding it, for example, or scrub its icon.

After processing all the text files in a project, NTK looks for a variable with the name RemoveScript. If it finds one, NTK uses it to build a remove script, which it places in the part frame in a slot named RemoveScript. Only some part types use remove scripts. (The part types are described in “Output Options” beginning on page 4-47.)

You can also optionally define a **deletion script** that’s executed only when the application’s icon is scrubbed—not when the card containing it is removed—as described in “Accessing the Part Frame” beginning on page 4-40.

The remove script is a function with one argument. In the case of the an application, the argument is the part frame:

```
RemoveScript (partFrame)
```

<i>partFrame</i>	The part frame for the application. Because the application has been removed, the theForm slot contains an invalid reference.
------------------	---

The following remove script, for example, removes an application’s registry with the Find and Intelligent Assist services.

Managing and Building a Project

```
RemoveScript := func(partFrame)
begin
    UnRegFindApps(kAppSymbol);
    unRegTaskTemplate(partFrame.result);
end;
```

In the case of an auto part, the argument to the remove script is the remove frame that was passed to the install script:

```
RemoveScript(removeFrame)
```

*removeFrame*            The remove frame, which contains at least one slot, which contains a copy of the remove script. The install script can add other slots to the remove frame.

An auto part must have both an install script and a remove script. “Auto Parts” beginning on page 4-49 describes auto parts.

Constants and Variables

NTK defines a number of constants and variables that you can use to access files and templates and to check the status of build options.

Table 4-1 lists the constants NTK defines before and during a build.

**Table 4-1**      Build constants defined by NTK

Constant	Value
home	The path name of the folder containing the open project file
kAppName	The application name you specify through the Application/Book section of the Application Settings dialog box
kAppString	The application symbol, which you specify through the Application/Book section of the Application Settings dialog box, stored as a string instead of as a symbol

## Managing and Building a Project

**Table 4-1** Build constants defined by NTK

Constant	Value
<code>kAppSymbol</code>	The application symbol you specify through the Application/Book section of the Application Settings dialog box
<code>kDebugOn</code>	True if Compile for Debugging is checked in the Project Settings dialog box
<code>kIgnoreNativeKeyword</code>	True if Ignore Native Keyword is checked in the Project Settings dialog box
<code>kPackageName</code>	The package name you specify through the Package Settings dialog box
<code>kProfileOn</code>	True if Compile for Profiling is checked in the Project Settings dialog box
<code>language</code>	The Language string specified through the Project Settings dialog box
<code>layout_filename</code>	A reference to the view hierarchy of the processed layout file named <i>filename</i>
<code>streamFile_filename</code>	A reference to the contents of a processed stream file named <i>filename</i> .

The `home` constant lets you reach a file in the same folder as the project file without specifying the entire path name. For example:

```
LoadDataFile(home & "data", myClass);
```

This statement loads the data file named `data` in the same directory as the open project file.

The constants `kDebugOn`, `kIgnoreNativeKeyword`, and `kProfileOn` let you check the status of compiler options during a build, so you can leave debugging and profiling code in place in your source code.

This statement, for example, prints a message to the Inspector window only if the option Compile for Debugging is enabled:

## Managing and Building a Project

```
if kDebugEnabled then Print("Executing this code");
```

The compiler removes simple conditional statements that always evaluate to `nil`. The example here leaves no trace in the output whenever `kDebugEnabled` is `nil`.

When NTK finishes processing a layout file, it creates a constant named `layout_filename`, which references the view hierarchy defined by that file.

The function `GetLayout`, described in “Compile-Time Functions” beginning on page 4-36, returns a reference to a view hierarchy. It is the preferred way to access an external layout file.

When NTK finishes processing a print format layout file, it creates a variable named `printFormat_filename`, which also references the view hierarchy defined by that file. This variable is redundant with the `layout_filename` constant; it remains for compatibility with earlier releases.

When NTK finishes processing an object stream file, it creates a constant named `streamFile_filename`, which references the contents of the stream file.

## Compile-Time Functions

---

NTK supplies a few compile-time functions that address the mechanics of building software. You can use these functions to

- define and use compile-time constants (`DefineGlobalConstant`, `UndefineGlobalConstant`, and `IsGlobalConstant`)
- access the templates that result from processed layout files (`GetLayout`)
- set and retrieve slots in the part frame (`SetPartFrameSlot` and `GetPartFrameSlot`)
- process a text file that’s not included in the project (`Load`)
- read a Newton object stream file (`ReadStreamFile`)

## Managing and Building a Project

## Defining Global Constants

---

You can use the compile-time function `DefineGlobalConstant` in a text file to create constants and initialize them with arbitrary expressions. You use `UndefineGlobalConstant` and `IsGlobalConstant` to undefine and test for global constants.

### DefineGlobalConstant

---

`DefineGlobalConstant` (*symbol*, *expr*)

*symbol*                      A symbol that names the value.

*expr*                         An expression that defines the value of the symbol.

The `DefineGlobalConstant` function creates a constant referenced by the specified symbol and with the specified expression value.

Use the `NewtonScript` constant syntax instead of creating a constant with `DefineGlobalConstant` whenever possible. You must use `DefineGlobalConstant` instead of the constant syntax to set a constant equal to a function.

You can use `DefineGlobalConstant` to make compile-time values available at run time without adding a slot to the application base view. You could incorporate your own named 'BMP' file, for example, by calling `GetBMPAsBits` in a definition:

```
DefineGlobalConstant('kWren,GetBMPAsBits("Wren", true));
```

You could access the bits from within the application templates by referencing the `kWren` constant. Suppose, for example, you're using a resource to draw an image when a button is tapped. You can send the `CopyBits` message, documented in the book *Newton Programmer's Guide*, in the button's `viewClickScript` method:

## Managing and Building a Project

```
:CopyBits(kWren, 5, 5, modeMask);
```

Functions you define with `DefineGlobalConstant` can be called with the NewtonScript call syntax or the `Apply` function. You could, for example, define a function with the symbol `kFunction`:

```
DefineGlobalConstant('kFunction, func(x,y) x + y);
```

You could then call the function within your application, without regard to inheritance, with

```
call kFunction with (2,40);
```

Functions you create with `DefineGlobalConstant` must be self-contained, that is, they must not depend on the view context.

The `DefineGlobalConstant` function accepts any valid expression that can be evaluated at compile time.

The `DefineGlobalConstant` function replaces the obsolete function `DefConst`.

## UndefineGlobalConstant

---

```
UndefineGlobalConstant(symbol)
```

*symbol*                      A symbol that names the constant.

The function `UndefineGlobalConstant` removes a global constant.

You use `UndefineGlobalConstant` to remove constants you've created with `DefineGlobalConstant`. This line, for example, removes the global constant with the symbol `kWren`:

```
UndefineGlobalConstant('kWren);
```

`UndefineGlobalConstant` always return `nil`.



## Managing and Building a Project

**IsGlobalConstant**

---

`IsGlobalConstant (symbol)`*symbol*                      A symbol that names the constant.

The function `IsGlobalConstant` reports whether a global constant with the specified name exists; it returns `true` if the constant is defined, `nil` if it isn't.

**Accessing Processed Templates**

---

You can use the compile-time function `GetLayout` to reference the frame containing a processed layout file.

**GetLayout**

---

`GetLayout (filename)`*filename*                      A string containing the filename of a layout file.

The `GetLayout` function returns a reference to the view hierarchy that resulted from the processing of the specified layout file. You use it to incorporate templates from external layout files.

To add items at the top of a find slip, for example, you place in the application base view a function that supplies the item templates, which you lay out in a separate file and incorporate with a function slot something like this:

```
myApp.FindSlipAdditions := func()
begin
return GetLayout("myFindSlipAdditions");
end;
```

You can also use `GetLayout` to place in your application a reference to a non-view object, such as the routing format frames required for sending data.

## Managing and Building a Project

You can use the `GetLayout` function in conjunction with the `.form` command in the Book Maker application to incorporate the layout files for any small, application-like elements included in your book files. The `.form` command, documented in the manual that accompanies the Book Maker application, requires the height and width of the layout base view, which you can get from the `viewBounds` slot for the view as displayed in the browser.

If the specified layout file hasn't been processed, the `GetLayout` function generates a compile-time error. The `GetLayout` function therefore provides earlier detection of unprocessed files than the `layout_filename` constant, which doesn't raise an error until the compiled code is executed.

## Accessing the Part Frame

---

You can use the `SetPartFrameSlot` function to add a slot to the part frame that's constructed during a build. You can use the `GetPartFrameSlot` function to retrieve the contents of slots added with `SetPartFrameSlot`.

You can use the `SetPartFrameSlot` function to define a **deletion script**—that is, a block of code that's executed when the icon for the package containing a part is scrubbed on the Newton. The deletion script is a function contained in the part frame in a slot with the symbol `deletionScript`. For example:

```
SetPartFrameSlot('deletionScript', func()
begin
foreach store in GetStores() do
    if s:HasSoup(kSoupName) then
        GetSoup(kSoupName):RemoveFromStoreXmit(kAppSymbol);
end);
```

## Managing and Building a Project

**SetPartFrameSlot**

---

`SetPartFrameSlot(slot, value)`*slot*                      A symbol for the slot to be added.*value*                      The value of the new slot.

The `SetPartFrameSlot` function adds a slot with the specified symbol and value to the part frame. If the slot already exists, `SetPartFrameSlot` changes its value.

If you specify a slot symbol that's also used by NTK, your definition is overridden during construction of the final part frame. You can't therefore use `SetPartFrameSlot` to establish an installation or removal script, for example, or to define the `theForm` slot.

**GetPartFrameSlot**

---

`GetPartFrameSlot(slot)`*slot*                      A symbol for the slot whose value you want.

The `GetPartFrameSlot` function returns the value of the specified slot in the part frame. Because NTK defines special slots like the install script and the remove script at the end of the build, you can't use `GetPartFrameSlot` to access those slots.

**Accessing Files That Aren't in the Project**

---

You can use the compile-time function `Load` to incorporate text files that aren't listed in your project.

You can use the `ReadStreamFile` function to read an object stream file—that is, a file in Newton Streamed Object Format.

## Managing and Building a Project

**Load**

---

`Load (pathname)`

*pathname*                      A string containing the path name of the file to be processed.

The `Load` function compiles and executes the contents of a NewtonScript file with the specified path name.

Placing files directly in the project list—instead of using `Load`—makes them accessible to the NTK Search and Find commands and is the preferred way to incorporate text files into a project.

**ReadStreamFile**

---

`ReadStreamFile (pathname)`

*pathname*                      A string containing the path name of the object stream file.

The `ReadStreamFile` function returns the object written in the specified stream file.

You can create object stream files in NTK, as described in “Output Options” beginning on page 4-47.

As an alternative to the `ReadStreamFile` function, you can add a stream file directly to your project and then access it with the constant `streamFile_filename`, which NTK defines when it processes the file. “Stream Files” beginning on page 4-50 describes how NTK processes stream files.

## Managing and Building a Project

Project-Build Function Summary

---

```

DefineGlobalConstant(symbol, expr)
UndefineGlobalConstant(symbol)
IsGlobalConstant(symbol)
GetLayout(filename)
SetPartFrameSlot(slot, value)
GetPartFrameSlot(slot)
Load(pathname)
ReadStreamFile(pathname)

```

Build Options

---

This section provides more details about the build options available through the dialog boxes described in “Establishing Settings and Preferences” beginning on page 4-6.

Compiling Native Code

---

NTK can produce not only byte code to be processed by the Newton interpreter but also native ARM code—that is, machine code to be executed directly by the Newton’s ARM chip.

Native code executes significantly faster, but it occupies much more space in memory. For compatibility with possible future models that don’t use the ARM chip, NTK produces both byte code and native code when it compiles a function into native code.

You can mark an individual function for native compiling by constructing it with the `func native` syntax:

```
func native (paramList) expression
```

You can invoke options in the Project Settings dialog box that cause NTK to ignore the `func native` syntax or to suppress the byte code when compiling native code.

Compiling functions into native code is described more fully in “Compiling Functions for Speed” beginning on page 8-10.

## Managing and Building a Project

## Embedding Debugging Information

---

You can specify a build with embedded debugging support through the Project Settings item in the Project menu.

When Compile for Debugging is enabled, the compiler

- adds a slot named `debug` to each view that you name through Template Info in the Browser menu. The value of the `debug` slot is the view's name. If you create your own `debug` slot for a view, however, NTK does not override that definition.
- adds to each NewtonScript function it compiles a slot named `DebuggerInfo` that contains either an integer or an array of debugging information. This information is used by the debugging functions described in Chapter 7, "Extended Debugging Functions."
- skips the step of combining objects, described in the following section.

You can check the value of the `kDebugOn` constant to provisionally compile your own debugging code only when Compile for Debugging is enabled, as illustrated in "Constants and Variables" beginning on page 4-34.

## Combining Objects

---

To reduce application size, NTK combines objects as a final step in the build process—that is, if two objects are identical, NTK combines them and references the single object wherever either object is used. If the string "New" appears in the text slot for two different buttons, for example, NTK creates a single text string object and references it in both button templates.

NTK combines objects only in frame-based part types (that is, not in store parts or stream files). Combining objects usually reduces package size by 10–20%. The main side effect is that combined objects are less likely to be stored near the code that references them.

The impact on performance is variable. The Newton OS pages package data in to system memory as the data is needed. Because objects might be further from the objects that reference them, more segments of a package might be paged in during execution. That probability is offset by the likelihood that there will be fewer pages overall.

## Managing and Building a Project

Be careful when using the functions that destructively modify strings; they can affect more than intended when used on shared objects. The most dangerous case is the empty string. Consider, for example, this frame:

```
constant kTemplate := {slot1: "", slot2: ""};
```

At the end of the build, NTK combines the two empty strings in the template into a single object. Suppose you create an instance of the frame at run time with `DeepClone` and then use `StrMunger` to alter the value of the instance:

```
local instance := DeepClone(kTemplate);
StrMunger(instance.slot1, 0, nil, "foo", 0, nil);
```

`DeepClone` creates a new writable string referenced in two places, and then `StrMunger` destructively modifies that string. Evaluating `instance.slot2` yields "foo". You can avoid the problem in this example by using `nil` instead of the empty string:

```
constant kTemplate := {slot1: nil, slot2: nil};
```

NTK normally combines objects in production builds, that is, builds where `Compile for Debugging` is not enabled. You can suppress the combining of objects in a production build by creating a global variable named `consolidateObjectsAfterBuilding` and setting it to `nil`.

## Profiling

---

NTK includes a profiling tool that keeps statistics on an application while it's executing on the Newton device. You can specify a build with profiling support through the `Project` tab found in the `Settings` dialog box in the `Project` menu.

To collect profiling statistics, you embed profiling code in your application and then build the application with `Compile for Profiling` enabled. When this option is enabled, the compiler assigns each function in the application a unique identifier that it maps back to the source code, and it recognizes the calls that turn profiling on and off during execution.

Chapter 8, "Tuning Performance," describes the profiling tool in detail.

## Managing and Building a Project

## Establishing a Local Language

You can specify through Project Settings a language string that the `LocObj` function uses to find localized versions of strings and other objects that change when a piece of software is compiled for use in a specific country or region.

The `LocObj` function takes two parameters: an object and a path name to an alternative object. If the language setting for a build is English, then `LocObj` returns the embedded object. If you set the localization string to any other value, `LocObj` looks for the object in the place specified by the language string together with the embedded path name.

If, for example, you display a message while searching for an object, you can set up the message for any language by wrapping the string in the `LocObj` function:

```
msg := LocObj("Searching for ^0...", 'find.searchfor)
```

The path name identifies a frame of localization data you establish—most likely in a text file—with the `SetLocalizationFrame` function:

```
SetLocalizationFrame({
    Swedish: {
        find: {
            searchFor:
                "Söker efter ^0...", // "Searching for ^0..."
            . . .}},
    French: {
        find: {
            searchFor:
                "Recherche dans ^0...", // "Searching for ^0..."
            . . .}}
});
```

When the Language setting in the Project Settings dialog box is English, NTK uses the string included in the code itself ("Searching for *name*"). When the



## Managing and Building a Project

Language setting is Swedish, NTK looks for the string contained in the slot `Swedish.find.searchFor` in the language frame.

The localization chapter in *Newton Programmer's Guide* describes the `LocObj` function.

## Output Options

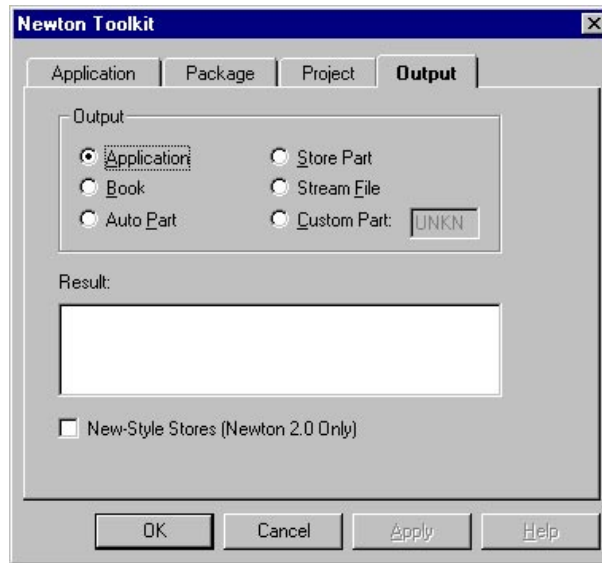
---

You can use NTK to create either a package file or an object stream file. A package file holds software to be installed on the Newton device. An object stream file holds a hierarchy of NewtonScript frames encoded in Newton Streamed Object Format. The NTK platform files, among other things, are stored in Newton Streamed Object Format.

A package file consists of a header containing package information and one or more parts containing code and data. Each build produces one new part. You can incorporate additional parts in a package by putting their package files in the project, as described in “Package Files” on page 4-5.

You specify the type of the new part through the Output tab of the Settings dialog box, illustrated in Figure 4-13 and described in “Output Settings” beginning on page 4-16.

## Managing and Building a Project

**Figure 4-13** Output Settings

When you download a package to the Newton device—or when you insert a PCMCIA card or otherwise add a software package—the Newton system software installs the package by reading the header information and dispatching the parts to the appropriate handlers.

NTK places an application, book, or auto part into a single **part frame** that holds the slots appropriate to a part of that type.

You can add your own slots to the part frame with the `SetPartFrameSlot` function, described in “Accessing the Part Frame” beginning on page 4-40.

### Application Parts

NTK stores an application in a part of type `form`. You assemble an application from NTK layout files plus any text, bitmap or sound file, or other files you need. You must designate one layout file as the main layout

## Managing and Building a Project

file; it holds the application base view, which is the view that's created when you start up the application.

After processing all files in the project, NTK looks for global variables with the special names `installScript` and `removeScript`. If it finds one or both, it uses them to create install and remove scripts, which it places in the part frame. The sections "Install Scripts" and "Remove Scripts," beginning on page 4-32, describe install and remove scripts in more detail.

NTK also looks for a global variable with the name `partFrame`. If one exists, and if it contains a frame, then the slots in that frame are copied to the application's part frame. The approved way to add slots to the part frame, however, is with the `SetPartFrameSlot` function, described on page 4-41.

---

### Book Parts

NTK stores an interactive book in a part of type `book`. You build a book from text files created by the Book Maker application, plus any layout files, bitmap or sound files, or other files containing book elements.

A book doesn't have a main layout file, and it doesn't use install and remove scripts.

---

### Auto Parts

An auto part holds software that isn't represented by an icon in the Extras drawer. You can use an auto part to add a panel to the Prefs roll, for example, or supply an application with data. You build an auto part from one or more text files, plus any layout files, bitmap or sound files, or other files you've used.

An auto part has no application base view, no application name, and no application symbol. Its part type is `auto`.

You can place your own data in an auto part frame by defining a global variable with the special name `partData`. After processing all files in the project, NTK looks for a variable named `partData`; if it finds one, it places its value in the part frame in a slot with the name `partData`. NTK also recognizes the global variables `installScript` and `removeScript`. An auto part must have an install script.

## Managing and Building a Project

---

**Parts in Auto-Remove Packages**

---

You can activate Auto Remove Package in Package Settings to specify that parts in the package are to be removed automatically immediately after installation.

When it encounters an auto-remove package, the Newton system software executes the install script for each part in the package and then removes the software, without executing a remove script or a deletion script. The only recommended constituent of an auto-remove package is a single auto part.

---

**Store Parts**

---

A store part holds a read-only store containing one or more soups. You create store parts from one or more text files plus any other files you've used to store the data. A store part is not a frames part; it has no slots for install and remove scripts, a part name, or an application symbol. A store part is of type soup.

When building a store part, NTK creates a global variable named `theStore`, which contains a store. Code that executes during the build can write data to the store; at the end of the build, NTK creates a part of type soup that contains all data written to the store during the build.

For more information about creating and using store parts, see *Newton Programmer's Guide*.

---

**Stream Files**

---

A stream file holds a hierarchy of NewtonScript frames in Newton Streamed Object Format (NSOF).

You can use stream files to incorporate into a project code or data that's already been processed. You could place a large data structure into a stream file, for example, and then incorporate it into a new project without rebuilding the structure every time you build the project.

NTK builds a stream file essentially the same way it builds any other kind of project: it processes the source files in order and places the results in a new

## Managing and Building a Project

file. You can then place that file in the project, where it will be processed in order during the build.

When it encounters a stream file during the build, NTK does two things:

- It looks for an `install` slot in the frame at the top of the hierarchy, and if it finds one, sends the `install` message to the frame. This allows your stream file to define its own global functions or other objects.
- It creates a constant named `streamFile_filename`, which references the contents of the stream file. You can then use this constant to incorporate the contents of the stream file into your software.

The NSOF specification is available under some restrictions from Apple—to request the specification, send mail to [tools@newton.apple.com](mailto:tools@newton.apple.com).

When you choose Stream File in Output Settings, NTK displays a Result field. You must enter in the Result field an expression—typically a global variable—that evaluates to the top-level frame of the output file.

### Custom Parts

---

You can use NTK to create parts of any type, including dictionary parts and font parts, by choosing Custom Part in Output Settings and entering a four-character type code in the type field.

When it builds an application, a book, or an auto part, NTK builds a part frame with the slots appropriate for a part of that type. When it builds a custom part, NTK makes no assumptions about what slots to add to the output frame. When you're assembling a custom part, you must build your own part frame.

When you choose Custom part in Output Settings, NTK displays a Result field, as illustrated in Figure 4-14. You must enter in the Result field an expression—typically a global variable—that evaluates to the top-level frame of the output.

## Managing and Building a Project

**Figure 4-14** Custom part settings

## Build Sequences

---

This section summarizes the guidelines for ordering files in the project window and describes how NTK processes a layout file.

### Building a Project

---

NTK processes the files in a project in the sequence you establish through the project window. NTK requires that you group files by type, in this order:

- bitmap or sound files
- package files
- text files, layout files, and object stream files
- package files

When it can identify the file types, NTK enforces the order as you add or rearrange files through the project window.

To reorder files in the project, select one file at a time and

- choose Process Earlier from the Project menu or press Ctrl-Up Arrow to move that file closer to the beginning of the build or
- choose Process Later or from the Project menu or press or Ctrl-Down Arrow to move it closer to the end of the build.

NTK builds one new part out of the text and layout files in the project. If you include package files in the project, NTK places them before or after the new part, depending on where you placed them in the project.

## Managing and Building a Project

The next section, “Processing a Template,” describes how NTK processes layout files. “Stream Files” beginning on page 4-50 describes how NTK processes stream files.

## Processing a Template

---

When NTK processes a layout file, it starts by processing the template for the layout view, the parent of all other templates in the file. In the course of processing a template, NTK processes all of its children. Therefore, the processing of a layout file begins and ends with the layout view, which is the first template in a file whose processing is started and the last template whose processing is completed.

After it finishes processing the template for the layout view, NTK creates the constant `layout_filename`, which contains a reference to the view hierarchy defined by that file.

NTK processes each template in a layout in four steps:

1. NTK looks for a `_proto` or `viewClass` slot. If the template is based on an unprocessed user proto, NTK displays an alert and halts the build.
2. NTK compiles and executes the code in the `beforeScript` slot, if it's present.

Memory objects created in the before script are available to evaluate slots in the template and its descendants.

The `beforeScript` slot exists only during the processing of the current template; the `beforeScript` slot does not appear in the frame that results from the processing of a template.

The `beforeScript` and `afterScript` slots let you execute code specific to a template during the build. You can use the `beforeScript` slot to define functions and data that will be available before the processing of a single template, the same way you use text files to define functions and data that will be available during the rest of the build.

3. NTK builds the template:
  - ☐ NTK creates the template and adds the `_proto` or `viewClass` slot.
  - ☐ NTK creates the slots altered or added through the browser.

## Managing and Building a Project

As it creates each slot, NTK establishes the slot's value. When it creates an evaluate or script slot, NTK compiles and executes any NewtonScript code in the slot.

- NTK creates the `stepChildren` slot (described in the “Views” chapter of the book *Newton Programmer's Guide*).
  - NTK processes the template's children, adding each child to the `stepChildren` array as it's created.
4. NTK compiles and executes the code in the `afterScript` slot, if it's present.

The template is available to the after script through the variable `thisView`, which is a reference to the view template. You can use `thisView` to add slots or change the value of existing slots. This code in an `afterScript` slot, for example, would conditionally add an extra slot named `debugInfo` and place data in it:

```
if kDebugOn then thisView.debugInfo := data to be saved;
```

This code creates the extra slot only when Compile for Debugging is enabled. “Embedding Debugging Information” beginning on page 4-44 documents the Compile for Debugging option.

**Warning**

The `thisView` variable gives your after script access to any slot in a view. Use it carefully. ▲

The code in the `afterScript` slot is not part of the final application.

You can create `beforeScript` and `afterScript` slots for any view through the New Slot item in the Browser menu, documented in “Adding Slots” beginning on page 5-18. Create the slots as evaluate slots with the names `beforeScript` and `afterScript`.

## Error Messages

---

NTK displays its own error messages with explanatory text in the Inspector window.



## Managing and Building a Project

When NTK encounters errors in the code its compiling or receives errors from other sources, it displays an error message with the error number. You can look up error numbers in the “Errors” appendix in the book *Newton Programmer’s Guide*.

If you’ve installed the Newton application Exception Printer, which is shipped with NTK, the Newton itself displays more information about errors that arise during execution.

## CHAPTER 4

### Managing and Building a Project

# Laying Out and Editing Views

---

You use NTK's graphical editor to lay out your application's views, and you use the browser to add the code that defines how the views look and act. This chapter describes how you use the graphical editor and the browser.

You can adjust some features of the editor and browser through the Toolkit Preferences menu items, described in "Layout Preferences" on page 4-24 and "Browser Preferences" on page 4-25.

## Laying Out Views

---

You use NTK's graphical editor to lay out views in a window that represents the Newton screen.

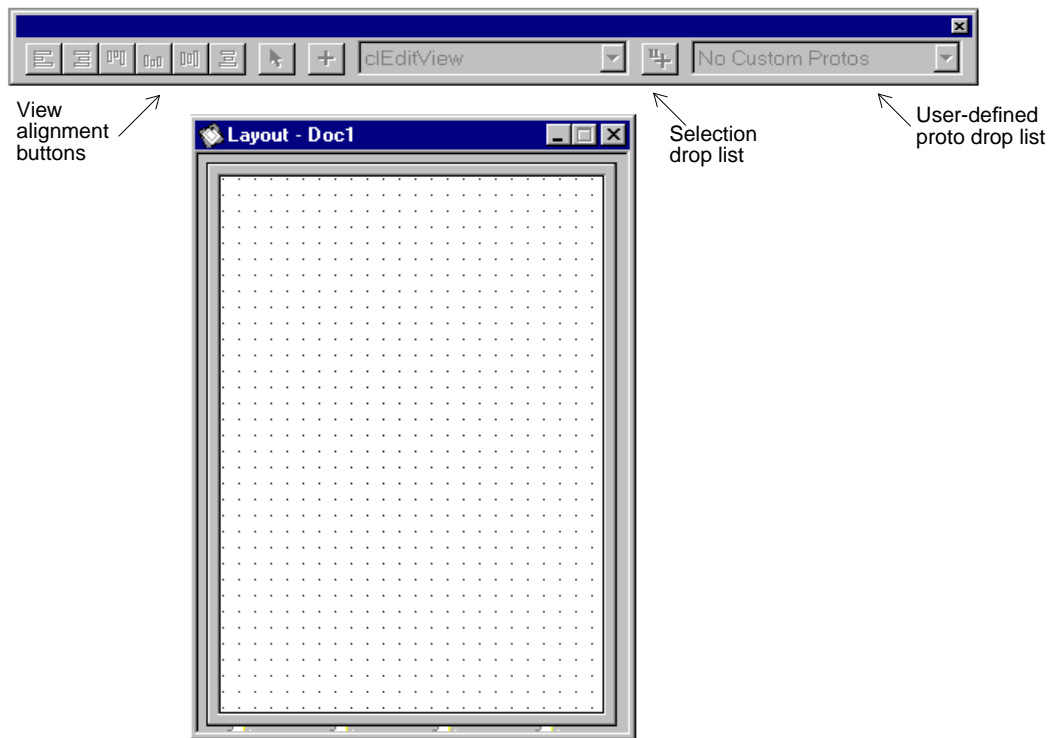
Figure 5-1 illustrates the MessagePad layout window and the toolbar of components, which NTK displays when you choose New Layout from the File menu or open a layout window. You can open a layout window by

## Laying Out and Editing Views

selecting a layout file in the project window and choosing Open Layout from the Windows menu.

When a project is open, the size of the layout window and the selection of components on the toolbar depend on the platform you've chosen through the Settings items in the Project menu. If no project is open, the size of the window depends on the screen size set through the Layout Preferences item in the Layout menu; the composition of the toolbar depends on the platform file of the last project that was open.

**Figure 5-1** Layout window and toolbar



## Laying Out and Editing Views

You lay out views by choosing a proto or view class from one of the drop lists and then drawing a view in the layout window, as described in “Drawing a View” beginning on page 5-4.

The graphical editor saves the views you lay out in a **layout file**, which contains a hierarchy of templates. Each layout file must have a single **layout view**, which contains all other views in the layout. The layout view can contain any number of child views, which themselves can contain any number of child views, and so on.

The view that opens when a user taps an application’s icon is the **application base view**. The application base view—which is the layout view for the application’s main layout file—is the ancestor of all other views in the application.

The Selection drop list lets you add these elements:

- view classes, the basic building blocks of view templates  
The view class `clParagraphView`, for example, is the generic text view, used for static or editable text. The view classes are built into the Newton.
- system protos constructed from the view classes  
The system protos, also built into the Newton, provide ready-to-use elements like radio buttons and slide controls.
- linked subviews, an NTK device for bringing into the hierarchy views laid out in separate layout files  
You can lay out your application in modules and then link the files together through linked subviews.

The User proto drop list lets you add user protos, that is, protos you define yourself. You can base your protos on view classes, system protos, or other user protos.

The *Newton Programmer’s Guide* describes the view classes and system protos. This chapter describes linked subviews in “Linking Multiple Layouts” beginning on page 5-14.

The view alignment buttons align selected views as illustrated on the buttons.

## Drawing, Resizing, and Moving Views

---

You use the mouse and a drop list of templates to add views.

### Drawing a View

---

To add a view to your application, you select the template you want from the appropriate drop list and then draw out the view in the layout window.

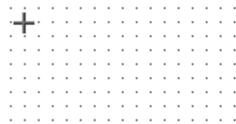
You select a template by choosing an item from one of the two drop lists found on the toolbar. The Selection drop list contains the view classes and proto templates built into the Newton ROM. The User proto drop list contains proto templates you've created and added to the current project. You activate the selection drop list by clicking the Selection button (the large button with an arrow on it); you activate the User proto drop list by clicking the User button (the button with the letter U on it).

Once you've activated a component, move the cursor to the layout window.

To draw a view:

1. Place the tip of the arrow cursor where you want any corner of the view to appear.
2. Press and hold down the mouse button.

The cursor changes to a crosshair.



3. Hold down the mouse button while you drag the cursor to the opposite corner.

## Laying Out and Editing Views

4. When the view is the size and shape you want, release the mouse button.

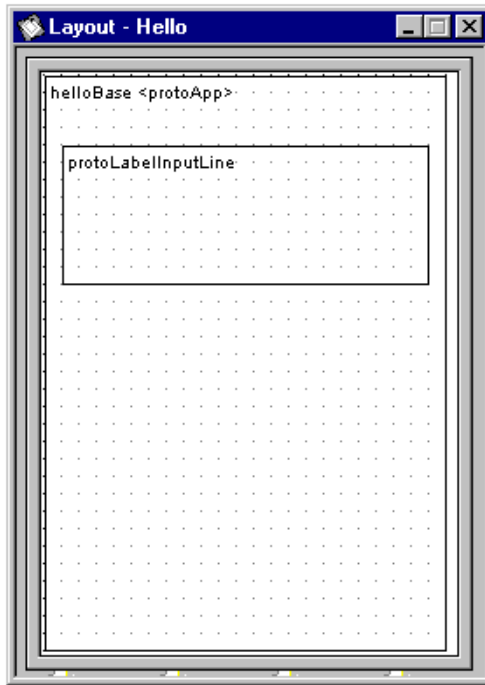


The rectangle you've defined on the screen determines the location of the view as stored in the `viewBounds` slot in the frame's template. You can anticipate different platforms by adjusting the `viewBounds` slot when a view is instantiated, as described in the "Views" chapter in *Newton Programmer's Guide*.

Figure 5-2 illustrates a layout window with a layout view and one child view in place.

## Laying Out and Editing Views

**Figure 5-2** A layout window with the layout view and one child view in place



The label in the upper-left corner of the view shows the view class or proto template on which the view is based. After you've named a view, its name appears as well.

Selection marks appear at the drawing corners of the selected view—`protoLabelInputLine` in Figure 5-2. The selected view is the target of whatever view-editing instructions you make through the mouse or keyboard. You select a view by clicking on it. To select multiple views, hold down the Ctrl key while clicking in the layout window.



## Laying Out and Editing Views

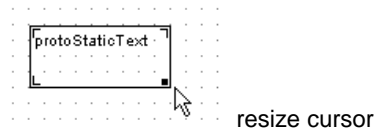
## Resizing a View

---

You can resize a view with either the mouse or the keyboard.

To resize a view with the mouse:

1. Select the view.
2. Place the cursor on the bottom-right corner of the view. When the cursor is placed for resizing, it changes to a two-headed arrow.



3. Press and hold down the mouse button while you drag the corner.
4. When the view is the size and shape you want, release the mouse button.

You can select and resize multiple views at once. If you simply resize multiple views with the resize cursor, NTK resizes the views proportionally, so that the selected views retain their relative sizes. If you hold down the Ctrl key while resizing multiple views, NTK resizes all the views by the same absolute amount, that is, the same number of pixels.

To resize a view with the keyboard:

1. Select the view.
2. Hold down the Ctrl key while pressing one of the arrow keys.

The Right-arrow key enlarges the view by moving the right edge one pixel to the right. The Left-arrow key shrinks the view by moving the right edge one pixel to the left. The Down-arrow enlarges the view by moving the bottom edge one pixel down. The Up-arrow shrinks the view by moving the bottom edge one pixel up.

To change the size of a view by five pixels at a time, hold down both the Shift key and the Ctrl key while pressing an arrow key. You can set the numbers of pixels views are resized by an arrow key alone and by Shift-Arrow key through the Toolkit Preferences item in the Edit menu, described on page 4-19.

## Laying Out and Editing Views

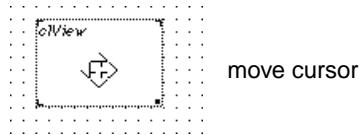
## Moving a View

---

To move a view with the mouse:

1. Place the cursor anywhere on the view (except the bottom-right corner) and press the mouse button.

The cursor changes to the shape seen below.



2. Hold the mouse button while you drag the view, and release the button when the view is in the position you want.

If you press and hold the Shift key while moving a view, NTK constrains the movement to either the vertical or the horizontal axis, depending on which direction you move in first.

To move a view with the keyboard:

1. Select the view.
2. Press any of the arrow keys.

The arrow keys move the view one pixel in the direction of the arrow. You can move the view five pixels at a time by holding down the Shift key while you press the arrow key.

You can set the number of pixels a view is moved by the arrow key alone and by Shift-Arrow key through the Toolkit Preferences item in the Edit menu, described on page 4-24.

You can select and move multiple views at once.

## Aligning Views

---

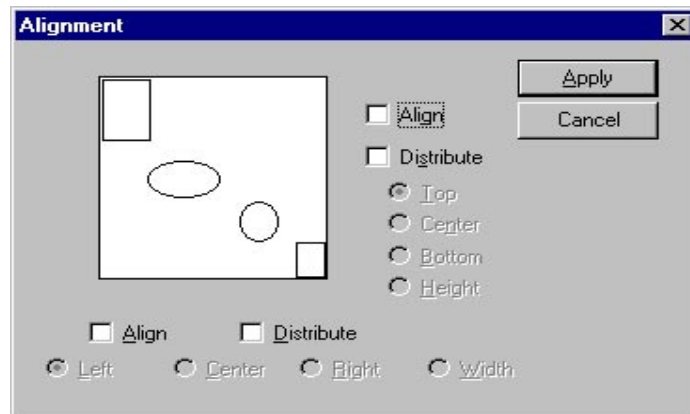
You can align the sides or centers of two or more views by selecting the views and clicking one of the alignment buttons on the toolbar.



## Laying Out and Editing Views

You can perform more sophisticated alignments through the Alignment and Align items in the Layout menu. Choosing Alignment displays the dialog box illustrated in Figure 5-3.

**Figure 5-3** The Alignment dialog box



As you select various alignment options, the objects in the sample rectangle move to show the effect. Once you've set up your alignment rules through the Alignment dialog box, click Apply to apply them to the selected views. You can later choose Align from the Layout menu to apply the current alignment rules to the selected views.

Often it's more appropriate to handle alignment programmatically through the parent- and sibling-relative options of the view system.

### Vertical Spacing

The options to the right of the sample rectangle control vertical spacing. You can either align or distribute selected views during one alignment.

## Laying Out and Editing Views

You can align the tops, centers, or bottoms of selected views. When aligning tops and bottoms, NTK aligns all selected views to the top of the topmost view or the bottom of the bottommost view. When aligning centers, NTK centers all views over the line halfway between the top of the topmost view and the bottom of the bottommost view.

You can distribute the selected views so that the tops, centers, or bottoms are evenly spaced, or so that the distance is the same between the tops and bottoms of adjacent views.

### Horizontal Spacing

---

The options below the sample rectangle control the horizontal spacing. You can either align or distribute selected views during one alignment.

You can align the left sides, centers, or right sides of selected views. When aligning left and right sides, NTK aligns all views with the view furthest to the left or right, respectively. When aligning centers, NTK centers all views over the line halfway between the outer sides of the most distant views.

You can distribute the selected view so that the left sides, centers, or right sides are evenly spaced, or so that the distance between the edges of adjacent views is the same.

### Ordering Views

---

Views are drawn on the Newton screen in the order in which they appear in the drawing list. Views that appear later in the list can obscure views drawn earlier.

Within each sibling group, views are added to the drawing list in the order you lay them out in NTK's graphical editor. You can move a view one place ahead in the drawing list by selecting it and choosing Move Backward from the Layout menu. You can move a view one place back in the drawing list by choosing Move Forward. You can move a view behind all its siblings in the drawing list by choosing Move To Front, and you can move a view ahead of its siblings in the drawing list by choosing Move To Back.

## Laying Out and Editing Views

You can also reorder views by selecting them in the browser template list and pressing Ctrl-Up arrow (to move a view forward in the drawing list) or Ctrl-Down arrow (to move a view back in the drawing list).

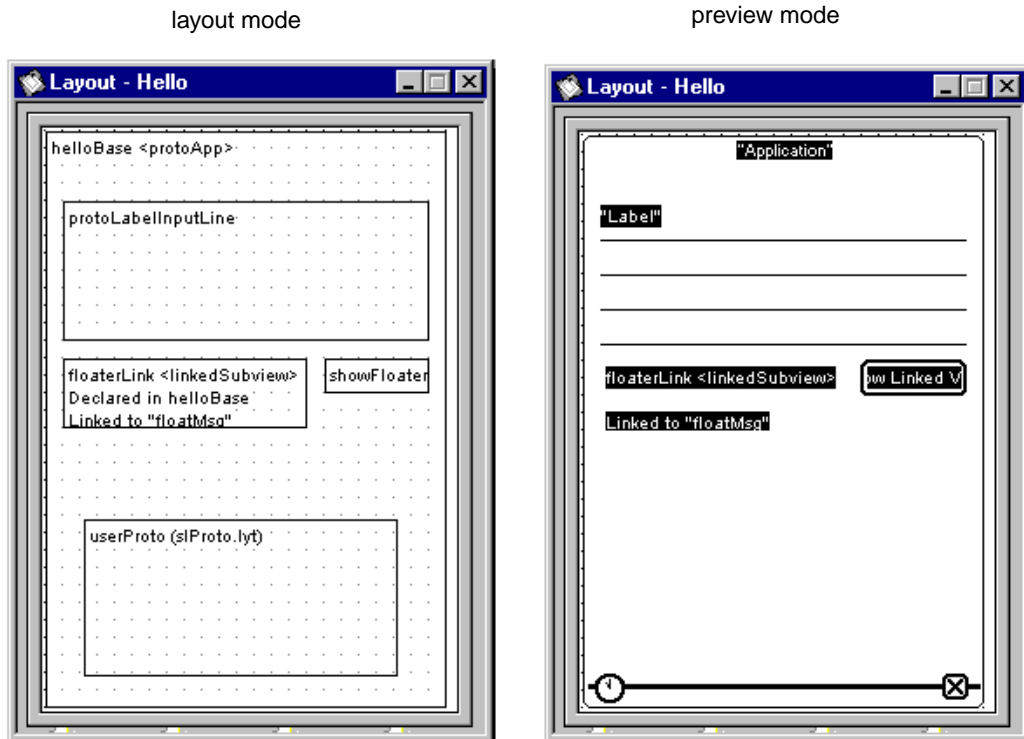
## Previewing

---

You ordinarily draw views with the graphical editor in layout mode, in which NTK displays the rectangular extents of the views and their names. You can see a closer approximation of how the views will look on the Newton screen by choosing Preview from the Layout menu.

Figure 5-4 illustrates a simple view in layout mode and preview mode.

## Laying Out and Editing Views

**Figure 5-4** The layout window in layout and preview modes

Views based on the most commonly used protos appear in preview mode much like they'll appear on the Newton screen. Text is displayed only in the default font, and a `protoStaticText` view can display no more than 255 characters. User protos are not displayed.

You can toggle between layout and preview modes by choosing Preview from the Layout menu or pressing Ctrl-Y.

Preview mode is fully implemented for the templates `protoApp`, `protoCheckBox`, `protoLabelInputLine`, `protoRadioButton`,

## Laying Out and Editing Views

protoStaticText, protoTextButton, protoPictureButton, protoSlider, clGaugeView, and protoRadioCluster.

## Naming and Declaring Views

---

You name and declare views through the Template Info item in the Browser menu.

You must supply unique names for

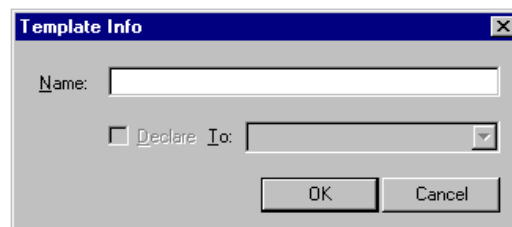
- views that declare themselves to other views and
- views to which other views declare themselves.

You should also name all views that you'll need to identify through the browser template list. You don't need to name all views—NTK identifies unnamed views with a label based on the name of the view's proto.

When you choose Template Info, NTK displays the dialog box illustrated in Figure 5-5.

---

**Figure 5-5** The Template Info dialog box, for naming and declaring views



You name a view by typing into the Name field. You activate the Declare To drop list by clicking its check box. Click the mouse button with the cursor on the menu to see a list of the view's named ancestors. Declare a view only to its immediate parent.

## Laying Out and Editing Views

You must name a view before you can declare it. Declaring a view places a slot for that view into the template in which you are declaring it, allowing symbolic access from the parent to the child. The “Views” chapter in *Newton Programmer’s Guide: System Software* contains a more complete discussion of declaring views.

## Linking Multiple Layouts

---

You can work on an application in separate layout files, each with its own local main view. You link layout files with the special-purpose linked subview element.

To link an external file to an application:

1. In either the main layout file or a layout file that’s linked to the main layout file, lay out a small reference view, using the `linkedSubview` item in the Selection drop list.
2. Link the linked subview to the external layout file by choosing Link Layout from the File menu.

The section “Adding a Linked Layout” beginning on page 3-16 illustrates how to link a separate layout file into an application.

If you make a link to a file that’s not already in the project, NTK automatically adds the file to the project. If you remove a linked file from a project (with the Remove File item in the Project menu), you also remove the information about links to that file. The external file must appear in the project list before the file that references it.

The linked subview is a placeholder in the parent view. When the parent template is processed, the templates in the linked layout file replace the linked subview template. The name of the linked subview, however, replaces the name of the layout view in the linked file.

To declare a view in the linked layout file to an ancestor in the other file, you must declare the placeholder view (the linked subview) in its parent file. Child views in the linked layout file then declare themselves to the layout

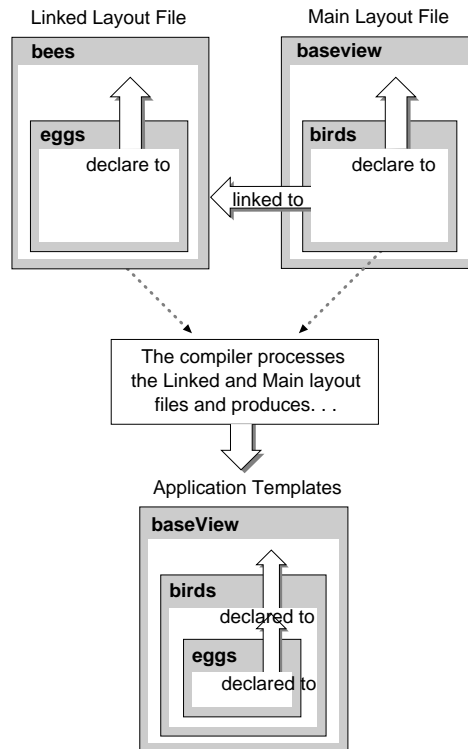


## Laying Out and Editing Views

view in the linked layout file; they send messages up the hierarchy by referencing the placeholder view.

For example, consider a placeholder view with the name `birds`, in the `baseView` template in the main layout file. The layout view in the linked layout file is named `bees`, and it has a child named `eggs`, as illustrated in Figure 5-6.

**Figure 5-6** Declaring views across linked layout files



## Laying Out and Editing Views

In this example, the view `birds` declares itself to `baseView`, and the view `eggs` declares itself to `bees`. To send a message to its parent, the view `eggs` sends the message to `birds`. To send a message to `eggs`, `baseView` sends the message to `birds.eggs`.

If you assign the same name to the placeholder view and the main layout view in the linked layout file, you don't need to remember which name to reference.

## Creating User Protos

---

You can use the New Proto Template item in the File menu to start your own proto layout. You save the layout in a separate file and add it to the project through the Project menu.

Once you've added your proto to the project, you can lay out views based on it by choosing it from the User proto drop list on the toolbar.

The tutorial section "Defining Your Own Proto" beginning on page 3-23 illustrates how to define and use your own proto template.

## Browsing and Editing Templates

---

You use the NTK browser and slot editors to program your templates.

### Browsing Templates

---

A browser window lets you examine the templates in a local view hierarchy and the slots within each template.

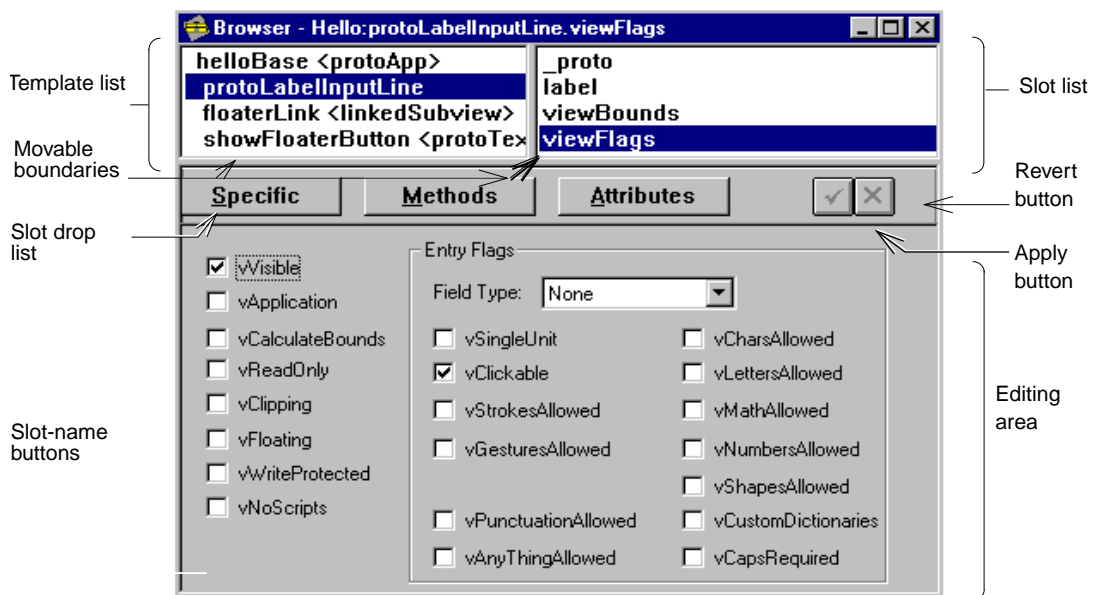
You open a browser window by opening a saved layout file or by selecting a view in a layout window and choosing New Browser from the Window menu. You can examine and edit slots in the template for the selected view or any of its descendants. If you want access to all of the templates in a layout file, choose New Browser with no view selected, with the layout view

## Laying Out and Editing Views

selected, or with the layout file selected in the project window. You can keep several browser windows open at once.

Figure 5-7 illustrates a browser window with the `viewFlags` slot open for editing.

**Figure 5-7** A browser window with the view flags slot open for editing



The template list in the top-left corner lists the templates in the view hierarchy. The slot list to the right lists the slots in the selected template. Highlighting around either the template list or the slot list shows which is active; you can change the selection in the active window by pressing the Up arrow and Down arrow keys.

## Laying Out and Editing Views

You open a slot for editing by clicking its name in the slot list. The browser then displays the slot's contents in the editing area in the lower part of the window. If you've changed the template selection since opening a slot, clicking on the slot-name button in the editing area restores the template and slot selections.

You can resize the three panes of the window by moving the boundaries with the mouse. You can customize the amount of information displayed and the text styles used in the template and slot lists through Browser Preferences, described in "Browser Preferences" beginning on page 4-25.

NTK provides different editors for different kinds of slots. Figure 5-7, for example, illustrates the view flags slot editor. "Editing Slots" beginning on page 5-20 describes the basic slot editors; Appendix E, "Slot Editors," lists the specialized slot editors

The Apply and Revert buttons allow you to apply or cancel any editing you've done to a slot since the last revert or apply. When you click Apply (or press Ctrl-E or choose Apply from the Browser menu), NTK places any outstanding changes into the slot. When it applies a change, NTK checks for syntax errors in NewtonScript code. It reports errors in the Inspector window but applies the changes in any case. NTK automatically applies changes to a slot when you

- open a different slot for editing
- close the browser window
- save the file in which the slot is stored
- build a package.

## Adding Slots

---

You can add any of the slots defined by the Newton system software through the three slot pop-up menus:

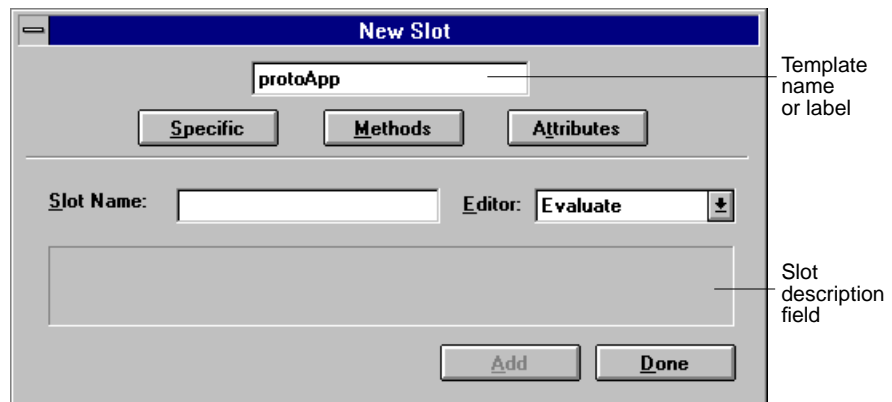
- Specific—lists the proto-specific slots in the proto template on which the selected template is based

## Laying Out and Editing Views

- **Methods**—lists the system-defined **methods**, that is, the code that executes when a view receives one of the system messages
- **Attributes**—lists **attributes**, that is, the view characteristics the Newton system software uses when displaying and manipulating views.

You add your own slots to a template through the dialog box illustrated in Figure 5-8, which NTK displays when you choose New Slot from the Browser menu.

**Figure 5-8** The New Slot dialog box



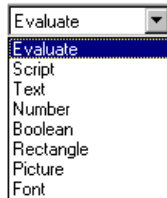
The name centered at the top of the window's content area (protoApp in this example) is the name of the template you're adding slots to or—if the template isn't named—a label based on the name of the template's proto or view class. You can add any of the system-defined slots through the proto pop-up menus, and you can add your own slot by typing the slot name into the Slot Name field.

Whenever the Slot Name field contains the name of a system-defined slot, the description field contains a brief description of the slot.

## Laying Out and Editing Views

When you're defining your own slots, you specify a slot type—which implies a slot editor—through the Editor drop list, illustrated in Figure 5-9. The slot types are documented in the following section.

**Figure 5-9** The Editor drop list in the New Slot dialog box



When you've established a slot name and editor, click Add. To dismiss the dialog box, click Done.

## Editing Slots

You use the basic slot editors listed in this section to edit slots of the types available through the New Slot dialog box. NTK also supplies specialized editors for editing various system-defined slots, listed in Appendix D, "Slot Editors."

An **evaluate slot** is a slot that's evaluated in place, that is, during the build when the code is compiled. You use evaluate slots to embed data that's available only during the project build into the templates that will be used on the Newton device. The value of the slot is set to the value returned by the last statement executed.

A **script slot** holds a function definition that's compiled during the build for execution at run time. NTK processes evaluate and script slots in exactly the same way: During the build, NTK first compiles the contents of the slot, then executes the resulting code with the NewtonScript interpreter, and finally sets the value of the slot to the value returned by the last statement executed.

## Laying Out and Editing Views

The result for an evaluate slot is a value. The result for a script slot is a function.

A **text slot** holds text. During the build, NTK places the specified text in a text string in the text slot.

Figure 5-9 illustrates the initial displays for evaluate, script, and text slots.

---

**Figure 5-10** Initial contents of evaluate, script, and text slots

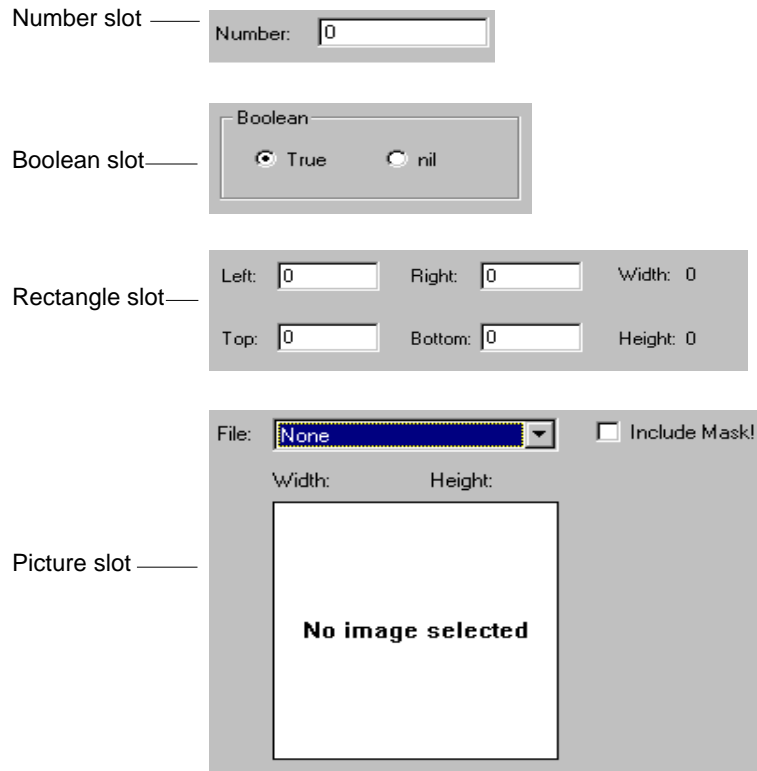


You edit evaluate, script, and text slots with the text editor described in “Editing Text” beginning on page 5-23.

If you delete the keywords `func()`, `begin`, and `end` from a script slot, it becomes equivalent to an evaluate slot; conversely, if you place a function in an evaluate slot, it’s equivalent to a script slot.

Number, Boolean, rectangle, and picture slots use editors tailored to their data. Figure 5-9 illustrates the initial display for these four slot types.

## Laying Out and Editing Views

**Figure 5-11** The number, Boolean, rectangle, and picture slot editors

A **number slot** can hold either an integer or a real number. If you enter an integer in the range -536,870,912 to 536,870,911, NTK stores it as type `integer`. If you enter an integer outside that range or a number containing a decimal point, NTK stores it as type `real`.

A **Boolean slot** can hold only the value `true` or `nil`.

The **rectangle slot** holds four integers. NTK automatically calculates the width and height based on the integers you supply. In your own rectangle



## Laying Out and Editing Views

slots, you can use the four integers however you want. In a `viewBounds` slot, the meanings of the four values depend on the value of the template's `viewJustify` slot, as documented in Appendix D, "Slot Editors."

A **picture slot** holds a 'BMP' file. You use the picture slot editor to specify a bitmap. The File drop list displays all bitmap files in the project. The selected picture appears in the rectangle to the right of the resource list. NTK displays the width and height in pixels.

A picture's mask is a parallel 'BMP' file that's used to display the image when it's selected. To supply your own mask, place it as a bitmap file in the project, with a trailing exclamation point on the file name and check Include Mask!. If the file is named *wave*, for example, the mask takes the name *wave!*. If you don't supply your own mask, NTK generates a simple one automatically from the original.

## Editing Text

---

You edit text in slots and text files with a text editor that follows the basic user interface conventions:

- The blinking cursor marks the current insertion point, that is, the place where keystrokes are inserted. You change the insertion point by moving the cursor with the mouse and clicking at the new insertion point.
- You select text by holding down the mouse button and dragging the cursor through the text to be selected. Double-clicking selects the word in which the cursor appears. Triple-clicking selects an entire line.
- The Cut, Copy, and Paste items in the Edit menu (and their keyboard equivalents: Ctrl-X, Ctrl-C, and Ctrl-V) delete selected text from the slot and place it on the clipboard, copy the selected text to the clipboard without deleting it, and paste the contents of the clipboard, respectively.
- Keystrokes replace selected text.

You can also navigate, select, and manipulate text with the arrow keys and keystroke combinations listed in Appendix A, "Keyboard Text-Editing Commands."

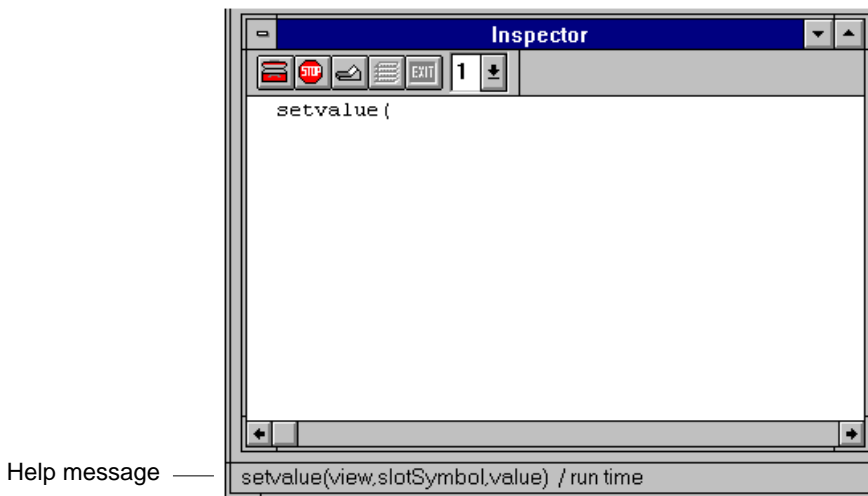
## Laying Out and Editing Views

As a help mechanism, NTK supplies argument information when you're entering global functions and common messages. When you type a left parenthesis after a token that's listed in the editor's internal database, NTK displays a help line in the status bar at the bottom of the NTK window. To see the arguments for the `setValue` function, for example, type

```
setValue(
```

The help message appears in a box in the status bar at the bottom of the NTK window, in either a browser or the Inspector window, as illustrated in Figure 5-12.

**Figure 5-12** The Inspector window with a help message displayed



The search and display are triggered by the typing of the parenthesis, not by the position of the cursor.

## Searching for Text in Files

---

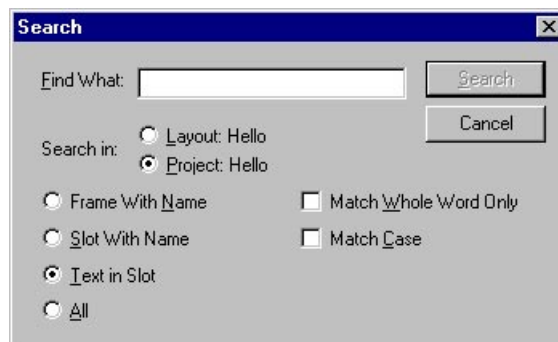
The Edit menu contains a number of items that let you search for strings in various situations.

## Searching Template Files

---

The Search item identifies all instances of a string in a template file or in all template files in a project. You specify the string and the search criteria through the dialog box illustrated in Figure 5-13.

**Figure 5-13** The Search dialog box



Select one of the radio buttons at the top of the window to specify the files to be searched.

In Layout (Ctrl-L)

Searches only the layout file associated with the active layout or browser window.

In Project (Ctrl-P)

Searches all layout and text files in the open project.

## Laying Out and Editing Views

Select one of the four radio buttons below the string field to specify the target of the search.

## Frame With Name (Ctrl-F)

Searches layout files for frames whose name contains the specified string.

## Slot With Name (Ctrl-S)

Searches layout files for slots whose name contains the specified string.

## Text In Slot (Ctrl-T)

Searches layout and text files for slots whose value contains the specified string.

## All (Ctrl-A)

Searches layout and text files for the specified string in frame names, slot names, or slot values.

You can limit the search by checking one or both of the boxes:

## Whole Word (Ctrl-W)

Finds only instances in which the specified string appears as a word, that is, in which the specified string is not embedded within other text.

## Case Sensitive (Ctrl-E)

Finds only strings that match the capitalization of the specified string.

When you click Search, NTK finds and lists all occurrences of the specified string. You can double-click any of the entries to open or activate a browser window with that entry selected. If NTK finds no instances of the string, it sounds the system beep.

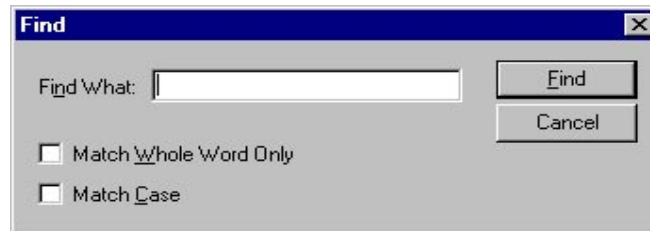
## Searching the Active Window

---

The Find and Find Next items search through text in the active window to find and select a specified string.

Find displays a dialog box, illustrated in Figure 5-14, in which you specify the string and search specifications.

## Laying Out and Editing Views

**Figure 5-14** The dialog for searching with Find

The Whole Word option finds only instances in which the specified string appears as a word, that is, in which the specified string is not embedded within other text. The Case Sensitive option finds only strings that match the capitalization of the specified string. You can toggle the checkboxes by pressing Alt-W and Alt-C on the keyboard for Whole Word and Case Sensitive, respectively.

When you click Find, NTK finds and selects the next occurrence of the specified string. If NTK finds no instances of the string, it sounds the system beep.

The Find Next item finds the next occurrence of the string last found through Find.

The Find and Find Next items are available when you're working in the Inspector window and when you're editing a text file or a slot that contains text.

## Finding Views in a Layout File

The Find Inherited item finds and selects in the layout file the view that contains a slot with the same name as the slot selected in the browser slot list.

The Find Inherited command looks first in the parent of the selected template. If it doesn't find the selected slot there, it continues up the parent hierarchy to the top level. When it finds a slot with the same name as the

## Laying Out and Editing Views

selected slot, NTK opens another browser window, with the slot and its template selected. If it doesn't find the slot in any template in the hierarchy, NTK sounds the system beep.

## Adding Non-View Objects

---

Although most Newton objects are views, you occasionally need a non-view object, like the format frame required for beaming frame data.

To create a non-view object that you can edit in the browser, place a simple view—such as a static text view—in a separate layout file. You can remove the unneeded slots—the `viewBounds` and `text` slots in the case of a static text view—by selecting and deleting them in the browser. You can add an `afterScript` slot that redefines the `_proto` slot (or removes the `_proto` slot and replaces it with a `viewClass` slot, if that's what you need).

This `afterScript` slot, for example, redefines the `_proto` slot to a format frame:

```
thisView._proto := protoFrameFormat
```

To remove frames in an `afterScript` slot, use the `RemoveSlot` function, documented in *Newton Programmer's Guide: System Software*.

Add the layout file to your project. You can access the processed templates with the `GetLayout` function, which is described in "GetLayout" beginning on page 4-39.

To create a non-view object without the browser, you can type the frame into a text file.

## Customizing the Text Editor

---

You can install your own keystroke definitions to the NTK text editor by adding them to the array `protoEditor.keys` through your global data file, which is described in "Global Data File" on page 4-29.

## Laying Out and Editing Views

The following line in your global data file, for example, causes selected code to be evaluated when you press the period key on the numeric keypad.

```
protoEditor:DefineKey({key: 65}, 'EvaluateSelection');
```

The EvaluateSelection method is built into the editor.

The following example defines a function upcaseSelection, which converts selected text to upper-case text, and ties the function to the key combination Ctrl-U.

```
protoEditor.upcaseSelection := func(off, len)
begin
    :ReplaceSelection(Upcase(:Selection()));
end;
protoEditor:DefineKey({key: $u, option: true},
    'upcaseSelection');
```

## Laying Out and Editing Views



# Debugging

---

This chapter explains how you can use NTK to debug software running on the Newton. This chapter describes

- the Inspector, a debugging window that lets you examine the Newton from the development system
- a collection of debugging functions you can issue interactively or embed in software under development
- a few common NewtonScript programming problems

## **WARNING**

The functions described in this chapter are for debugging purposes only. Do not include them in released products. ▲

The file NS Debug Tools.pkg, which is shipped with NTK, contains a collection of debugging functions that let you examine the execution environment in more detail. For a further discussion of debugging, see Chapter 7, “Extended Debugging Functions.”

NTK is also shipped with a collection of special-purpose debugging tools that you can install on the Newton. These tools are listed in Appendix E, “Newton Debugging Applications.”

## Debugging

## Compatibility

---

This chapter describes the debugging functions that are built into the Newton ROM and available through NTK. All but two of these functions are available on both the MessagePad and Newton 2.0 platforms.

This chapter describes the stack trace display on a Newton 2.0 PDA. The current function in a stack trace on a Newton MessagePad is at stack level 2.

The `GetSelfFromStack` and `GetLocalFromStack` functions described in this chapter are available only on the MessagePad platform. To examine the execution environment on a Newton 2.0 device, you must use the functions described in Chapter 7, “Extended Debugging Functions.”

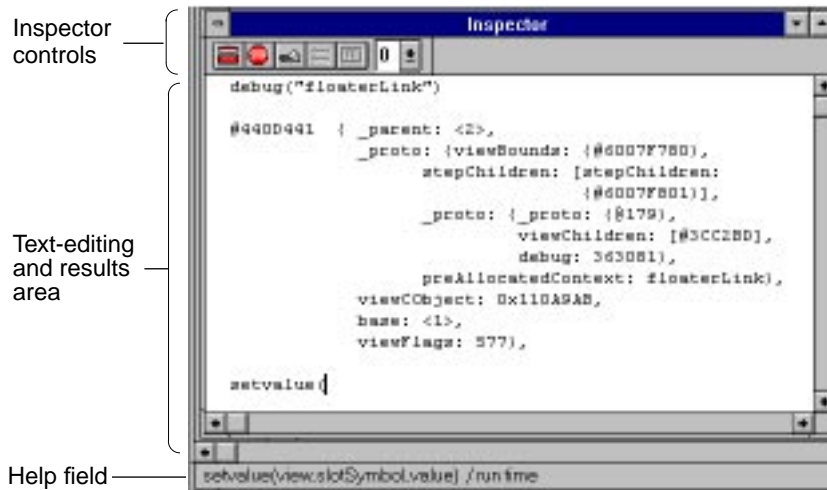
## The Inspector

---

The Inspector is a debugging window that lets you browse the Newton object storage system and execute NewtonScript code on the Newton. You make an Inspector connection through the Toolkit application running on a Newton attached to the development system by a serial cable, as described in Chapter 1, “Installation and Setup.”

You communicate with the Newton through the Inspector window, illustrated (with an open connection) in Figure 6-1.

## Debugging

**Figure 6-1** Inspector window

You use the buttons along the top of the Inspector window to open and close the connection with the Newton and to control the debugging environment, as described in “Using the Inspector” beginning on page 6-5.

You edit text in the Inspector window with the editor described in “Editing Text” beginning on page 5-23 and in Appendix A, “Keyboard Text-Editing Commands.”

The help field displays the parameters and return values of common functions and messages when you type a left parenthesis after a function or message name, as illustrated in Figure 6-1.

You execute code you’ve typed in the Inspector window by selecting the text and pressing Enter in the numeric keypad, or by pressing Ctrl-Enter on the keyboard. If you press Enter with no text selected, the Newton executes the current line.

## Debugging

**NOTE**

Pressing the Return key does not trigger an evaluation of text. ♦

After the Newton executes the selected code, the Inspector window displays the results of the last statement executed.

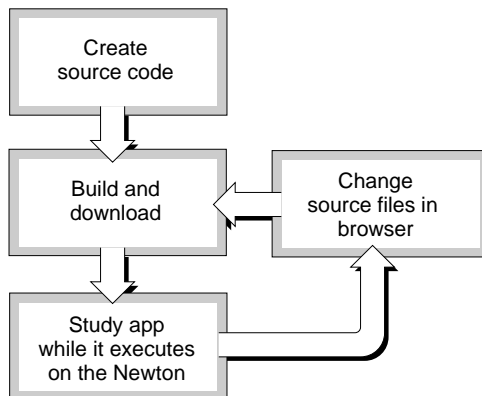
The Inspector window gives you access to the NewtonScript interpreter and the object store on the Newton. You can execute any valid NewtonScript code in the Inspector window, and you can examine the Newton with the functions described in this chapter.

The Inspector window also displays warnings and error messages that arise during execution, during a build, or when you apply changes in the browser.

You can save the contents of the Inspector window through the File menu, and you can open the Inspector window without connecting to the Newton by choosing Open Inspector from the Windows menu.

You can use the Inspector to study your program while it's executing and to test out proposed changes. You can then make changes in the source code, rebuild the package, and download the new version. Figure 6-2 illustrates the debugging cycle.

**Figure 6-2** The debugging cycle



## Debugging

## Using the Inspector

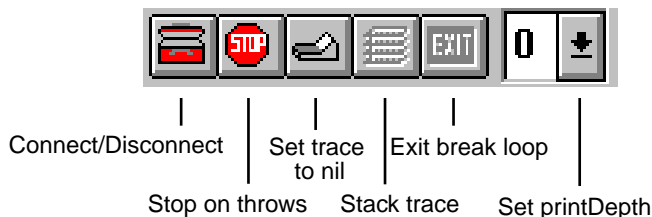
---

You can use an Inspector connection to

- examine and edit views and other objects
- trace the flow of execution
- collect performance statistics.
- study memory use
- examine your application's drawing efficiency

You interact with the Newton device by entering commands in the text-editing area and manipulating the Inspector controls, which are illustrated in Figure 6-3 and described in the following subsections.

**Figure 6-3** Inspector controls



The Print Depth drop list—to the right of the Inspector buttons—determines how many levels of the frame hierarchy are displayed in the Inspector window when you enter a command that displays frames. The Print Depth drop list sets the value of the `printDepth` variable, which is described in Table 6-1 on page 6-21.

## Making an Inspector Connection

---

You initiate an Inspector connection by clicking the Connect button or choosing Connect Inspector in the Windows menu. The Newton must be

## Debugging

attached to the development system by a serial cable, as described in Chapter 1, “Installation and Setup.”

You complete the connection from the Newton side by opening the Toolkit application and tapping Connect Inspector.

When you’re typing into the Inspector window, you’re communicating directly with the Newton, at what’s referred to as the **top level**. You can create objects and define global variables, which you can access by simply typing their names. Suppose, for example, you enter this text:

```
seaFrame := {name: "Pacific",
             color: "blue",
             size: "large"};
```

You can then display the frame by entering its name:

```
seaFrame;
```

The Inspector processes the statement and displays the results in two ways: a transient reference (a hexadecimal number preceded by a pound sign) and a textual representation:

```
#440B9C9 {name: "Pacific",
          color: "blue",
          size: "large"}
```

Applications that you build on the development system and download to the Newton are declared in the root view under their application symbol—there’s a slot in the root view whose name is the application symbol and whose value is the application base view. To reach objects defined in an application, you must find them within the hierarchy. To see whether an application with the signature `hello:TUT` is open, for example, you could test the `viewCObject` slot with this statement:

```
call kViewIsOpenFunc with (GetRoot().|hello:TUT|);
```

The rest of this chapter describes a number of functions that let you examine objects on the Newton device through an Inspector connection.

## Debugging

**WARNING**

The functions described in the rest of this chapter are for debugging purposes only. Do not include them in released products. You can place them in your source code conditionally, as described in “Constants and Variables” beginning on page 4-34. ▲

## Retrieving Views

---

In software that has been compiled for debugging, you can use the `Debug` function to retrieve a view made from a named template. This statement, for example, returns the view built from the template named `helloBase`.

```
Debug( "helloBase" );
```

The `Debug` function searches all templates on the Newton looking for a slot named `debug` whose value is a string that begins with the specified characters. When Compile for Debugging is enabled in Project Settings, NTK automatically creates a debug slot containing the template’s name in any named template in your project. You can also add a slot named `debug` to any frame in your application. NTK does not override the value you assign to a debug slot you create yourself.

If it finds a match, `Debug` returns the view and displays a text representation of it in the Inspector window:

```
#440E4B1  {_Parent: {#440ABB1},
           _proto: {#600828A1},
           viewCObject: 0x110AAB9,
           floaterLink: {#440FCA9},
           viewBounds: {#440FCF9},
           viewclipper: 17865641,
           base: <1>,
           viewFlags: 5}.
```

The values of the `printDepth` and `printLength` variables, described in Table 6-1 on page 6-21, control how much information the `Debug` function

## Debugging

displays. The example here shows the display when `printDepth` is set to 0 and `printLength` is set to `nil`.

## Displaying the View Hierarchy

You can display the hierarchy under a view with the DV function, which takes a view as its parameter. To display the hierarchy under the view based on the template named `helloBase`, for example, you would enter:

```
DV(Debug("helloBase"));
```

Figure 6-4 illustrates the output.

**Figure 6-4** The DV display

```
helloBase          #44100D9 [ 10,  4,230,320] 10000005 vVisible vApplication vHasChildrenHint
|180244            #4416681 [103,  2,137, 18] 40000003 vVisible vReadOnly
|362116            #4416751 [ 10,302,230,320] 50000001 vVisible vHasChildrenHint
|1291204           #44167B9 [ 14,302, 31,319] 60000201 vVisible vClickable vHasidlerHint
|46154304          #4416841 [211,304,224,317] 40000203 vVisible vReadOnly vClickable
|93132404          #4416881 [ 18, 52,226,140] 50000201 vVisible vClickable vHasChildrenHint
|12915956          #44168C9 [ 80, 52,226,140] 40003A01 vVisible vClickable vGesturesAllowed vCharsAllowed vNil
|15793920          #4416919 [ 18, 54, 80, 67] 40000203 vVisible vReadOnly vClickable
|showFloaterB      #44168E9 [132,150,224,170] 40000203 vVisible vReadOnly vClickable
|sliderHolder      #4416C01 [ 34,244,210,316] 50000001 vVisible vHasChildrenHint
|360448            #4416C69 [ 50,260,194,276] 40000201 vVisible vClickable
|outputView        #4416C19 [170,284,194,300] 40000003 vVisible vReadOnly
|15793920          #4416CD1 [ 98,284,162,300] 40000003 vVisible vReadOnly
|floaterLink       #4416649 [ 30,162,210,222] 10000041 vVisible vFloating vHasChildrenHint
|1451352           #4417001 [196,208,209,221] 40000203 vVisible vReadOnly vClickable
|15793920          #4417019 [ 38,178,214,218] 40000003 vVisible vReadOnly
#2                NIL
```

For the view and each of its children, the DV function displays:

- the name of the view or—if it's based on a proto template and is not otherwise named—the name of the view's proto encoded as an integer

### Note

If you install the Newton package `DebugHashToNames.pkg`, the Inspector can translate some of the integers to proto names. ♦

- an internal reference to the object, prefaced with the pound sign (#)



## Debugging

- the view's bounds (left, top, right, bottom) in global coordinates
- a hexadecimal number that is the value of the `viewFlags` slot
- a list of the view flags set for the view

The first view listed is the view specified as the parameter to `DV`. Child views appear with their own children beneath them, with vertical bars to indicate each view's position in the hierarchy.

Like the global function `GetView`, `DV` recognizes three special symbols for the *view* argument:

- The `'viewFrontMost'` symbol returns the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot.
- The `'viewFrontMostApp'` symbol returns the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot).
- The `'viewFrontKey'` symbol returns the view on the screen that currently accepts keystrokes.

These symbols are evaluated at run time.

## Displaying Values in the Inspector Window

---

You can display the value of objects in the Inspector window with the `Print`, `Write`, and `Display` functions, each of which takes an object and displays its value. You can display a hexadecimal string representation of a binary object with the `StrHexDump` function, described on page 6-25.

The `Print`, `Write`, and `Display` functions are similar to each other, but they follow different display conventions.

The `Print` function displays an object and adds a newline. It places quotation marks around strings and places a dollar sign in front of characters. The `Display` function places quotation marks around output and identifies characters but does not add a newline. The `Write` function adds no special marks or newlines.

## Debugging

Consider a button whose text slot contains the value "Show Print" and whose `buttonClickScript` slot contains this method:

```
func()
begin
    Print("Using the Print function");
    Print(text);
    Print("x");
    Print("\n");
    Print($x);
end
```

When you tap the button with an Inspector connection open, the Inspector window displays this text:

```
"Using the Print function"
"Show Print"
"x"
"
"
$x
```

Similarly, a button with the text slot "Show Display" and a similar `buttonClickScript` method produces this output:

```
"Using the Display function" "Show Display" "x" "
"$x
```

Finally, a button with a text slot containing "Show Write" and a similar `buttonClickScript` method produces this output:

```
Using the Write functionShow Writex
x
```

The `Print`, `Display`, and `Write` functions are useful for debugging, but they do nothing but waste time and space on a stand-alone Newton device.

## Debugging

Examining a Binary Object

---

You can use the `StrHexDump` function to retrieve a string that's the hexadecimal representation of a binary object. This example displays the object in the text slot in the view named `showFloaterButton`, with a space after every four bytes of output:

```
print(StrHexDump(Debug("showFloaterButton").text, 4));
```

The output is a hexadecimal representation of the text string, with a space after every four bytes:

```
"00530068 006F0077 0020004C 0069006E 006B0065 00640020
00560069 00650077 0000"
```

The `StrHexDump` function is described in the section “`StrHexDump`” on page 6-25.

Breaking

---

You can often examine problems more closely by putting the Newton device into a **break loop**, in which execution of the program is suspended and the Newton accepts input only from the Inspector window. While the Newton is in a break loop, you can examine the program stack, examine and edit objects on the Newton, and execute NewtonScript code.

You can set a fixed break point in your application by embedding the `BreakLoop` function in your source code:

```
If kDebugOn then BreakLoop();
```

You can also instruct the Newton device to enter a break loop when an exception is thrown by clicking Stop on Throws or setting the `breakOnThrows` variable. When `breakOnThrows` is set to a non-`nil` value, the NewtonScript interpreter reports each exception to the Inspector—before it searches for a NewtonScript exception handler—and then enters a break loop. This option allows you to examine the situation before the exception handlers are invoked.

## Debugging

If the Newton encounters another exception or otherwise executes the `BreakLoop` function when it's already in a break loop, it enters a subsidiary break loop. The Inspector reports the level of the break loop as the Newton enters and exits.

To emerge from a break loop, click the Exit Break Loop button or enter the `ExitBreakLoop` function:

```
ExitBreakLoop( );
```

You can raise your own exceptions and define exception handlers to modify the flow of execution. *The NewtonScript Programming Language* describes NewtonScript exception handling.

You can disable breaking for exceptions by setting `breakOnThrows` to `nil`.

When `breakOnThrows` is `nil`, the Inspector reports only exceptions that aren't handled.

You can use the extended debugging functions to

- set break points in code that's already compiled and downloaded
- examine the Newton more thoroughly from a break loop.

Chapter 7, "Extended Debugging Functions," contains a further discussion of break loops and a description of the extended debugging functions.

## Examining the Program Stack

---

While the Newton is in a break loop, you can examine the program stack by clicking the Stack Trace button, which executes the `StackTrace` function.

The Inspector displays a trace, which is a series of run-time stack frames. For each frame on the stack, the Inspector displays this information:

stack level	The number of the stack frame.
function name	The name of the receiver. This is generally the name of a method or of a global function. The value <code>nil</code> in this field represents a built-in NewtonScript function without a name.

## Debugging

program counter     The value of the NewtonScript program counter within that function.

Suppose, for example, you're using a method stored in a slot named `initVector`. You attempt to invoke the initialization routine by calling the method by the wrong name:

```
: initArray(vector, 25);
```

The application compiles, but when you execute it on the Newton device, the application throws an exception. To find out which function is executing when the exception is raised, you enable Stop on Throws and execute the application again. This time, when it reaches the exception, the Newton reports the problem in the Inspector window and enters a break loop.

```
Undefined method: InitArray
evt.ex.fr.intrp;type.ref.frame
-48809
```

Entering break loop: level 1

You click the Stack Trace button, and the Newton device displays a stack trace something like this:

```
Frame 2:functions.BreakLoop   -1
Frame 3:buttonClickScript     0
Frame 4:viewClickScript       10
```

The most recent record on the stack appears first. The stack trace display does not show the first two frames (that is, frames 0 and 1), which are created and used by the Inspector. The stack level for the first record in the display is therefore 2. The current function—that is, the function that was executing at the time of the break—is in frame 3. In the example here, the break occurred during execution of the `buttonClickScript` method, which was called by the `viewClickScript` method.

You can examine the execution environment more closely with the functions described in Chapter 7, "Extended Debugging Functions." If you're using a Newton MessagePad, you can examine the environment with the

## Debugging

`GetSelfFromStack` and `GetLocalFromStack` functions described in this chapter on page 6-27.

If you've installed the extended debugging functions on the Newton, the `StackTrace` function produces the display described in "NewtonScript Stacks" beginning on page 7-6.

## Tracing the Flow of Execution

---

You can instruct the Inspector to report the flow of execution by setting the value of the trace variable.

You can set the trace variable to `functions` to instruct the Inspector to trace every function call and message send:

```
trace := 'functions;
```

The sending of the `TrackHilite` message, for example, with trace set to `functions` might appear like this:

```
Sending TrackHilite(18070494) to #440F671
=> TRUE
```

The number in parentheses after the function name is the argument value to `TrackHilite`. The hexadecimal number preceded by the pound sign is a reference to the view to which the message was sent. The second line (which starts with `=>`) is the return value from the `TrackHilite` method.

Depending on the types of the arguments, the trace displays either a reference to or a textual representation of the value of each. The function trace of the `InitVector` method used in the previous section, for example, might look like this:

```
Sending InitVector(#44124A1, 25) to #44104C1
  Calling SetLength(#44124A1, 25)
  => #44124A1
  Calling -(25, 1)
  => 24
  Calling Random(0, 100)
```

## Debugging

```

=> 32
Calling setAref(#44124A1, 0, 32)
=> 32
Calling Random(0, 100)
=> 29
Calling setAref(#44124A1, 1, 29)
=> 29
. . .
Calling Random(0, 100)
=> 84
Calling setAref(#44124A1, 24, 84)
=> 84
=> NIL

```

Setting trace to the value true causes the Inspector to report every frame and variable access.

A full trace generates significantly more output than a function trace. The trace of the TrackHilite function with trace set to true, for example, looks something like this:

```

Sending TrackHilite(18070461) to #440C921
get #44046C9 / #477.penSoundEffects = TRUE
get #440C921.viewCObject = 17865612
get #440C921.viewFlags = 2563
set #440C921.viewFlags = 33556995
get #440C921 / #440C4B9.icon = #6008CE71
get #440C921 / #3027B1.viewJustify = 2
=> TRUE

```

With full tracing in effect, execution quickly outstrips the display.

To turn off tracing, click the Trace Off button or, if possible amidst the scrolling output, set trace to nil. Scrolling may continue for some time, as the Inspector displays accumulated data.

## Debugging

## Examining Memory Use

You can use the `Stats` function to find out how much free memory is available in the NewtonScript heap and how big the largest contiguous free area is. For example:

```
Stats();
Free: 59716, Largest: 59540
```

The NewtonScript heap is a reserved part of system memory from which space for all NewtonScript objects is allocated.

You can execute the `GC` function immediately before `Stats` to ensure that all unallocated space is consolidated before you retrieve the memory statistics.

You can use the `TrueSize` function to calculate how much space an individual object requires in the NewtonScript heap. `TrueSize` adds together the sizes of the object itself and all of the heap objects it points to.

The total does not include read-only objects, such as objects in ROM or in the package. The total also excludes memory elements that can be automatically purged when more memory is needed, such as cached objects.

The `TrueSize` function reports the total number of objects measured and a breakdown by object type, as illustrated in Figure 6-5.

**Figure 6-5** A `TrueSize` display

```
truesize(Debug("sliderholder"), nil);
```

objects	11	300
frame	7	192
Array	2	52
map	2	56

The three columns list the object type, the total number of objects of that type, and the total size of the objects. The first entry, `Objects`, lists the totals.

The `TrueSize` function can also list

- some or all individual objects that were included in the calculation or



## Debugging

- all objects within the target that point to a specified object.

The Newton memory-management software can't remove an object as long as another object contains a reference to it. The listing of objects that reference an object helps you find obsolete references.

You supply a filter parameter that either suppresses the object listing or specifies which objects to list. Figure 6-6 illustrates a listing of all objects measured.

**Figure 6-6** A TrueSize display with object list

```

trueSize(Debug("sliderHolder"), 'all');

objects          11          300
frame            7          192
Array            2           52
map              2           56
  44      300    frame
  36      36     map
  36      84     Array
  28      48     frame
  24      24     frame
  24      24     frame
  24      24     frame
  24      24     frame
  24      24     frame
  20      20     map
  16      16     Array
                                map
                                kids
                                lowerSlider
                                outputView
                                lowerDisplay
                                countDisplay
                                kids[1]
                                kids[4]
                                lowerSlider.map
                                collectAverages

```

The four columns in the object-by-object listing show the size of the object itself, the size of that object together with the objects it points to, the class of the object, and its path name.

The paths are not exact path expressions. Frame maps, for example, cannot normally be referenced from NewtonScript, but they appear in the object list. Child views—which are listed with `kids` in the path name—are constructed from the view system, not from the `viewChildren` or `stepChildren` slots.

The filter parameter can be any of these values:

`nil` Displays the summary of objects by type and the frame in which it collected the data, as illustrated in Figure 6-7.

## Debugging

**Figure 6-7** The TrueSize summary and result frame

```

trueSize(Debug("sliderholder"), nil);

objects          11          300
frame            7          192
Array            2           52
map              2           56
#44141E1 {objects: {count: 11, size: 300},
         binary: {count: 0, size: 0},
         frame: {count: 7, size: 192},
         Array: {count: 2, size: 52},
         string: {count: 0, size: 0},
         symbol: {count: 0, size: 0},
         bitmap: {count: 0, size: 0},
         shape: {count: 0, size: 0},
         map: {count: 2, size: 56},
         Real: {count: 0, size: 0},
         instructions: {count: 0, size: 0},
         capture: NIL,
         reference: NIL,
         filter: NIL}

```

'all	Displays the summary and all objects measured, sorted by the size of the objects exclusive of the objects they point to.
'allKids	Displays the summary and all objects measured, sorted by the size of the objects inclusive of the objects they point to.
<i>classSymbol</i>	Displays the summary and all measured objects of the specified class. You can specify any of the classes listed in the result frame in Figure 6-7.
<i>reference</i>	Displays the summary and all paths within the specified object that point to the object specified by the reference, as illustrated in Figure 6-8.

## Debugging

**Figure 6-8** A TrueSize listing of references

```
truesize(debug("sliderHolder"), debug("outputView"));

objects          7          168
frame            4          104
Array            1          24
map              2          40
```

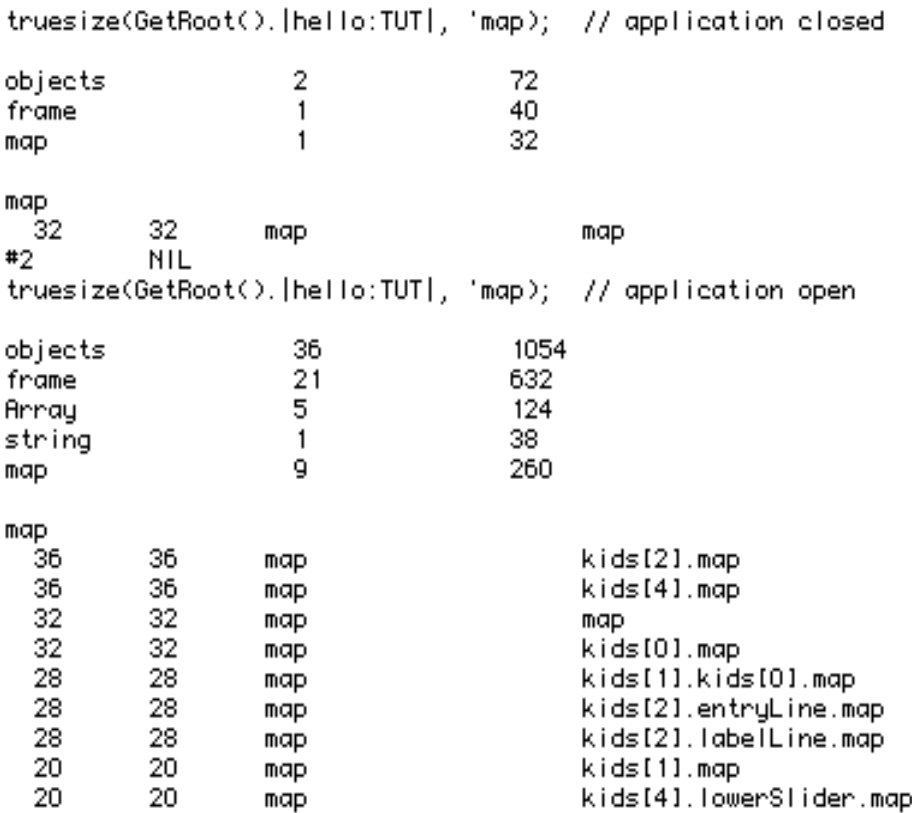
```
frame            outputView
frame            kids[1]
```

If you specify `nil` for the object to be searched, TrueSize searches the root view, the global variables, and the undo-buffer frame—that is, most of memory—for references to the object specified in the *filter* parameter.

You can use TrueSize to track the space used by an object over time. You can compare an application's needs immediately after a reset, for example, then while it's executing, and again after it exits. Figure 6-9 illustrates a TrueSize measurement over time of the tutorial application developed in Chapter 3, "A Quick Tour of NTK."

Debugging

**Figure 6-9** TrueSize measurements over time



Examining Drawing Efficiency

You can use the ViewAutopsy function to examine the efficiency of your application's drawing routines.

You can use ViewAutopsy in two different ways:

- outlining views      If you call ViewAutopsy with an argument of nil, the Newton toggles the outlining of views. When view outlining is in effect, the Newton displays the boundary

Debugging

of each view with a gray line. You can use this display to examine justification and view-layering problems.

slowing drawing      If you call `ViewAutopsy` with an integer argument, the Newton pauses for that number of tics after it draws each view. This option allows you to examine the sequence in which your application draws its views. You might find that views are redrawn, or possibly drawn and then obscured, before the display stabilizes. You can improve performance by eliminating unnecessary drawing.

To eliminate the delay, call `ViewAutopsy` with an argument of 0.

The `ViewAutopsy` function is described in “`ViewAutopsy`” on page 6-30.

## Debugging Variables

You can control how the Inspector operates by setting a number of variables, which are summarized in Table 6-1. “Using the Inspector” beginning on page 6-5 describes these variables as they arise.

**Table 6-1**      Debugging variables

Variable	Value	Effect
breakOnThrows	non-nil	The Inspector reports each exception and enters a break loop.
	nil	The Inspector reports only exceptions that are not handled by either the application or the Newton exception handlers.
trace	'functions	The Inspector displays tracing information for each function that's called and message that's sent.

# Debugging

**Table 6-1** Debugging variables (continued)

Variable	Value	Effect
	true	The Inspector displays tracing information for function calls and for variable and slot accesses.
	nil	The Inspector displays no tracing information.
printDepth	nil	A frame display shows all levels of the hierarchy.
	<i>n</i>	A frame display shows <i>n</i> levels of the hierarchy.
printLength	nil	A frame display shows all slots in the frame.
	<i>n</i>	A frame display shows the first <i>n</i> slots in the frame.

## Debugging Functions

This section describes the debugging functions that are built into the Newton system software. You can embed these functions in an application under development or call them interactively in the Inspector window.

### WARNING

The functions described in this chapter are for debugging purposes only. Do not include them in released products. You can place these functions in your source code conditionally, as described in “Constants and Variables” beginning on page 4-34. ▲

You can use the debugging functions to

- retrieve and display objects (Debug, DV, Display, Print, Write, and StrHexDump)
- enter and exit break loops and examine the program stack (BreakLoop, ExitBreakLoop, and StackTrace)

## Debugging

- examine memory use (Stats, TrueSize, and GC)
- slow down drawing and highlight view boundaries so you can examine your application's drawing efficiency (ViewAutopsy)

## Retrieving and Displaying Objects

---

You use the functions described in this section to retrieve objects and to print to the Inspector window.

### Debug

---

`Debug ( templateName )`

Returns the view whose template contains a slot named `debug` with a value that matches the string in the *templateName* parameter. NTK automatically creates a `debug` slot containing the name of any named slots on an application built with the Compile for Debugging option in effect.

*templateName*            The name of the template you want to examine, as a string.

This function scans all of the templates in the system and returns the view for the first match it finds. A template is considered a match if the initial characters in a slot named `debug` match the characters in *templateName*. If no match is found, the `Debug` function returns `nil`.

When `Debug` finds a match, it displays a textual representation of the view contents in the Inspector window and returns the view. The value of the `PrintDepth` variable, listed in Table 6-1 on page 6-21, controls the depth of the view display.

### DV

---

`DV ( view )`

Displays a view and its children in the Inspector window.

*view*                      The view object that you want to display.

The `DV` function always returns `nil`.

## Debugging

A quick way to display the contents of a view is to use the `Debug` function. To display the view made from a template named `helloBase`, for example, you would enter this text:

```
DV(Debug("helloBase"));
```

If a view is visible on the screen, `DV` produces a display of the view contents in the Inspector window, as described in “Displaying the View Hierarchy” beginning on page 6-8, and, if the application was built with `Compile` for `Debugging` in effect, flashes the view on the Newton screen. If the view is not visible, `DV` returns `nil`.

You can also specify one of three special symbols for the *view* argument:

- The `'viewFrontMost` symbol returns the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot
- The `'viewFrontMostApp` symbol returns the frontmost view on the screen that has the `vApplication` flag set in its `viewFlags` slot, but not including floating views (those with `vFloating` set in their `viewFlags` slot)
- The `'viewFrontKey` symbol returns the view on the screen that currently accepts keystrokes

## Print

---

```
Print(object)
```

Displays the value of *object* in the Inspector window.

*object*                      An object that you want displayed in the Inspector window.

The `Print` function lets you print any `NewtonScript` object. The `Print` function appends a newline character to its output, displays quotation marks around strings, and prefixes characters with `$`.

The `Print` function always returns `nil`.

For examples illustrating the `Print`, `Display`, and `Write` functions, see the section “Displaying Values in the Inspector Window” beginning on page 6-9.



## Debugging

**Display**

---

`Display(object)`

Displays the value of *object* in the Inspector window.

*object*                      An object that you want displayed in the Inspector window.

The `Display` function is exactly like the `Print` function except it does not append a newline character to its output.

**Write**

---

`Write(object)`

Displays the value of *object* in the Inspector window.

*object*                      An object that you want displayed in the Inspector window.

The `Write` function is exactly like the `Print` function except it does not append a newline character and does not display quotation marks around its text output.

**StrHexDump**

---

`StrHexDump(object, spaceInterval)`

Returns a hexadecimal string representing the value of the object.

*object*                      The binary object you want to examine.

*spaceInterval*              An integer specifying where to put spaces in the hex string output. To put spaces after every four bytes, for example, specify 4. For no spaces at all, specify 0.

You can use `StrHexDump` to examine the contents of a binary object.

**Note**

This function can return an extremely large string object, depending on the length of the binary object you specify. Use it carefully. ♦

## Debugging

## Using Break Loops

---

This section describes the functions you use to enter and exit break loops and to examine the program stack while in a break loop. You can examine the stack more closely with the extended debugging functions, described in Chapter 7, “Extended Debugging Functions.”

### BreakLoop

---

`BreakLoop( )`

Halts execution and allows you to examine the state of your application on the Newton device. You can also execute any valid NewtonScript code, including the functions built into the Newton, while in a break loop, as described in “Breaking” beginning on page 6-11.

If the Newton executes the `BreakLoop` function when it’s already in a break loop, it enters a subsidiary breakloop.

To exit a break loop, click the Exit Break Loop button or execute the `ExitBreakLoop` function.

### ExitBreakLoop

---

`ExitBreakLoop( )`

Exits a break loop.

When an Inspector connection is open, the Newton enters a break loop if

- it executes the `BreakLoop` function or
- an exception occurs while `BreakOnThrows` is `true`, as described in “Breaking” beginning on page 6-11.

If one of these conditions arises when the Newton is already in a break loop, it enters a subsidiary break loop. Execution of the `ExitBreakLoop` function exits only the current-level break loop. Program execution resumes when you exit the first-level break loop.

The `ExitBreakLoop` function always returns `nil`.

## Debugging

**StackTrace**

---

`StackTrace()`

Prints a stack trace in the Inspector window.

“Examining the Program Stack” beginning on page 6-12 illustrates a stack trace and describes its contents.

If you’ve installed the extended debugging functions, the `StackTrace` function displays the stack trace described in “NewtonScript Stacks” on page 7-6.

The `StackTrace` function always returns `nil`.

**GetLocalFromStack**

---

`GetLocalFromStack(level, symbol)`

Displays and returns the value of the local variable *symbol*.

*level*                      The number of the stack frame you want to examine.

*symbol*                    The symbol for the local variable you want to examine.

The first two entries on the stack—that is, levels 0 and 1—are used by the Inspector itself. To access the frame for the current function when the Newton is in a break loop, start at level 2.

The `GetLocalFromStack` function returns the value of the variable *symbol*.

**Note**

The `GetLocalFromStack` function is available only on the Newton MessagePad platform. ♦

**GetSelfFromStack**

---

`GetSelfFromStack(level)`

Returns the function at the stack frame level specified by the *level* parameter.

*level*                      The number of the stack frame you want to examine.

## Debugging

The first two entries on the stack—that is, levels 0 and 1—are used by the Inspector itself. To access the frame for the current function when the Newton is in a break loop, start at level 2.

**Note**

The `GetSelfFromStack` function is available only on the Newton MessagePad platform.

## Examining Memory Use

---

You use the functions described in this section to examine memory use on the Newton device and to force a garbage collection.

**Stats**

`Stats()`

Returns the amount of free memory in the NewtonScript heap and displays the amount of free memory and the size of the largest area of free memory.

The `Stats` function returns the amount of free memory in bytes. You can call GC first to ensure that any space occupied by unreferenced objects has been reclaimed.

**TrueSize**

`TrueSize(object, filter)`

Measures the total RAM requirements of an object by adding together its size and the sizes of all objects it points to. The total does not include read-only objects, such as objects in ROM or in the package.

*object*

A reference to the object to be measured.

If you pass a value of `nil`, `TrueSize` looks at the root frame, the global variables, and the undo-buffer frame. You use this option when looking for references to an

## Debugging

	object, as described in the description of the <i>filter</i> parameter.
<i>filter</i>	A filter that controls what data is collected and displayed.
<i>nil</i>	Displays the summary of objects by type and the frame in which the data was collected.
<i>'all</i>	Displays the summary and a list of all objects measured, sorted by the size of the objects exclusive of the objects they point to.
<i>'allKids</i>	Displays the summary and a list of all objects measured, sorted by the size of the objects inclusive of the objects they point to.
<i>classSymbol</i>	Displays the summary and all objects of the specified class.
<i>reference</i>	Displays the summary and all paths within the specified object that point to the specified reference.  To look for the reference throughout most of memory, pass a value of <i>nil</i> for the <i>object</i> parameter.

The `TrueSize` function summarizes the number and kinds of objects measured and collects specific data about some or all of them, as described in “Examining Memory Use” beginning on page 6-16.

**GC**


---

`GC ( )`

Forces a garbage collection in the NewtonScript frames heap, a reserved area of system memory from which the system allocates space for all NewtonScript objects.

The `GC` function frees all allocated objects that are no longer referenced. The Newton system software automatically performs a garbage collection when

## Debugging

memory is needed. You can call `GC` to ensure that unallocated space is consolidated before you call the `Stats` or `TrueSize` functions.

The `GC` function always returns `nil`.

## Examining Drawing Efficiency

---

You use the `ViewAutopsy` function to slow down drawing and highlight views so you can examine the efficiency of your application's drawing.

### ViewAutopsy

---

`ViewAutopsy(functionSpec)`

Provides two ways to examine how views are drawn. Supply a value of `nil` to turn on and off the outlining of views, in which the boundary of each view is marked by a gray line. Supply an integer to specify a pause (in ticks) after each view is drawn.

<i>functionSpec</i>	A value that specifies which drawing option you're manipulating:
<i>nil</i>	<p>Toggles view outlining.</p> <p>This option affects both the Newton screen and printed output. Use it for debugging justification and view-layering problems.</p>
<i>integer</i>	<p>Forces a pause for the specified number of ticks after each view is drawn.</p> <p>This option allows you to examine the drawing of views, so you can eliminate unnecessary redrawing.</p> <p>A value of 0 turns off the delay option with no effect on outlining.</p>

## Debugging

## Debugging Function Summary

---

### Retrieving and Displaying Objects

---

`Debug(templateName)`

`DV(view)`

`Print(object)`

`Display(object)`

`Write(object)`

### Using Break Loops

---

`BreakLoop()`

`ExitBreakLoop()`

`StackTrace()`

`GetLocalFromStack(level, symbol)`

`GetSelfFromStack(level)`

### Examining Memory Use

---

`Stats()`

`TrueSize(object, filter)`

`GC()`

### Examining Drawing Efficiency

---

`ViewAutopsy(functionSpec)`

## Newton Programming Problems and Tips

---

This section describes several common Newton programming problems and provides some programming tips.

This section addresses

- setting a slot in the wrong frame because of the inheritance rules
- forgetting to set the function value before exiting the function
- producing memory problems due to unused frame references
- generating unexpected comparison results when a value is `nil`
- generating errors when using `nil` in an expression
- trying to resize a read-only object
- drawing text that is not appearing on the screen
- attempting to print from within communications code
- using global variables to examine exceptions
- accessing the built-in error codes and messages

### Common Programming Problems

---

This section describes a number of common NewtonScript programming problems.

#### Setting the Wrong Slot Value

---

If you make an assignment to a slot that doesn't exist, NewtonScript automatically creates the slot in the current view; you may have intended to change a slot value elsewhere.



## Debugging

If the slot you are setting already exists in the view, your assignment works properly. For example, the assignment to the slot named `time` in the following example always works as expected.

```
myTemplate := {
    viewClass: clView,
    viewBounds: SetBounds(0,0, screenWidth, screenHeight),
    viewFlags: vApplication,
    debug: "myTemplate",
    time: 0,
    viewSetupDoneScript: func()
        time := Time()
    ...
}
```

If you make an assignment to a slot that doesn't exist, NewtonScript creates it as a local variable within the scope of the method only. If the slot exists only in the parent of your view, its value is set in the parent view.

Suppose, for example, that in the view in this example there were no slot named `time`, but the parent view did include a slot named `time`. Then, the assignment

```
time := Time()
```

would assign the value of the `Time` function to that slot in the parent view.

You can use the special pseudo-variable `self` to make sure that the slot is created in your view if it does not yet exist there. The value of `self` is always the current receiver. In the following example, a slot named `time` is created in the `myTemplate` view when the assignment statement is executed.

```
myTemplate := {
    viewClass: clView,
    viewBounds: SetBounds(0,0, screenWidth, screenHeight),
    viewFlags: vApplication,
    debug: "myTemplate",
```

## Debugging

```

    viewSetupDoneScript: func()
        self.time := Time()
    ...
}

```

Using `self` ensures that the slot is created if it does not exist in the current receiver.

Similarly, if you want to be sure that you are accessing or creating a slot in the parent of your view, use the `Parent` method, as shown here:

```
self:Parent().time := Time()
```

For more details on how inheritance affects setting slot values, see *The NewtonScript Programming Language*.

## Failing to Set a Return Value

---

Every function in NewtonScript returns a value, whether or not you explicitly assign one. If you use the return value, make sure that all pathways through your function establish one.

## Producing Memory Problems With Unused Frame References

---

If you maintain a reference to a child view, the Newton object system retains the child view. If you keep references to child views that are no longer needed, your application might run out of memory and display an exception.

The solution is to clean up (set to `nil`) your child view references when you are done with the views. This allows the Newton object system to reclaim the memory used by the view frame.

## Generating Unexpected Comparison Results With `nil` Values

---

If NewtonScript can't find a slot in a frame, it considers the slot's value to be `nil`. This assumption can mask mistyped slot names and produce misleading results.

Consider, for example, this function:

## Debugging

```
MyCompareFrame: func(frame1, frame2)
    begin
        if (frame1.date = frame2.date) and
            (frame1.month = frame2.month) then
            true;
        end
    end
```

If you made this call,

```
MyCompareFrame(frame1, frame2)
```

to compare the following two frames,

```
frame1 := {
    day: 5,           // note day instead of date
    month: 12,
};

frame2 := {
    day: 3,           // note day instead of date
    month: 12,
};
```

the result would be `true`. Neither `frame1` nor `frame2` contains a slot named `date`, which is what the `MyCompareFrame` function is comparing. The statement

```
frame1.date = frame2.date
```

evaluates to

```
nil = nil
```

which causes the function to return `true`.

## Debugging

Using nil in Expressions

---

A non-numeric value in a mathematical expression generates an error. For example, the following `buttonClickScript` method generates an exception:

```
func()
begin
    local index:=0;
    local str:="myString";
    print("The button has been clicked");
    index := index + StrPos(str, "xyz", 0);
    print("Reached the end of the button click script");
end
```

If you use this method, you'll see the following output in the Inspector window:

```
"The button has been clicked"
Exception |evt.ex.fr.type;type.ref.frame|: [-48404]
Expected a number. Got: {value: NIL}
```

You get this output because the `StrPos` function does not find the substring "xyz" and thus returns nil as its value.

You can also generate this error if you define a template that is based on a system proto and you forget to define one of the required numeric slots. If the value of one of these slots is used in a computation, then an exception is raised. For example, if you create a template based on the `protoInputLine` proto and do not define the `viewLineSpacing` slot, an error occurs because that value is used in the `viewDrawScript` method of the template.

Writing to a Read-Only Object

---

Templates are commonly read-only objects, and trying to alter one raises an error. This error arises in two common cases:

- trying to resize an array stored in a template

## Debugging

- trying to add children to a view whose `stepChildren` array is stored in a template

Suppose, for example, you create an evaluate slot with an array defined as its default value:

```
myArray: [1, 2, 3, 4]
```

When you instantiate the template at run time, a RAM-based view is created that inherits the read-only array. If you try to modify this array by adding elements to it, you receive a read-only error.

To add elements to the array, you need to clone it first to create a RAM-based copy:

```
if IsReadOnly(myArray) then myArray := Clone(myArray);
```

Cloning the array copies the read-only array into RAM, creates a new `myArray` slot in the RAM-based view, and puts a reference to the RAM copy of the array in that slot.

A read-only error can also occur with slots that contain frames and strings. The error is less common with strings, though, because they are usually replaced rather than changed in place.

You might also encounter problems with writing to a read-only object when you define a template that is composed of multiple sub-templates. The template contains a `stepChildren` array that is predefined. When you instantiate the template at run time, a RAM-based view is created that inherits the read-only array. If you want to add children to the view at run time, you need to clone the `stepChildren` array:

```
if not self.stepChildren then
    self.stepChildren := [];
else if IsReadOnly(stepChildren) then
    stepChildren := Clone(stepChildren);
AddArraySlot(stepChildren, newKidInTown);
```

## Debugging

## Text Is Not Drawing

---

If you've passed a valid string to the `MakeText` function, but the text does not appear on the screen, it's possible that the rectangle you've specified in the arguments to `MakeText` is too small. Try passing a larger rectangle to the `MakeText` function.

## Problems with Printing and Communications

---

Problems with the `Print`, `Write`, or `Display` functions are common in communications code, especially if the functions appear inside state frames. Check first for these two likely causes:

- The serial port is already in use for other communications. Only one communications channel can be open at any time.
- The `Print` statements are generating a lot of interrupts. This interferes with the serial line and causes hang-ups in the communications.

If you need to issue a quick message from within your communications code and need to avoid calling `Print`, `Write`, and `Display`, you can use the system notification facility instead. For example, you could display a quick message with the following code.

```
GetRoot():Notify(kNotifyAlert,
                EnsureInternal("My Comms App"),
                EnsureInternal("I'm low on memory"));
```

Another technique for reporting messages is to create an error array in the application base view. You can then add strings to this array from within your communications code, as shown here:

```
AddArraySlot(GetRoot().(kAppSymbol).DebugArray,
              "My Comms App:" && myDebugData);
```

## Debugging

## Programming Tips for Debugging

---

This section provides several suggestions to help you add debugging code to your applications.

### Using Global Variables to Examine Exceptions

---

If you are handling exceptions in your application, you can use the global variables shown in Table 6-2 to discover information about the exception. Note that the values of these variables are assigned by the top-level (system) exception handler, which means that you can use the values reliably only after the exception alert message has been displayed on the Newton screen.

**Table 6-2** Exception handling global variables

Variable	Description
<code>lastEx</code>	The string name of the most recent exception
<code>lastExError</code>	The integer error code of the most recent exception
<code>lastExMessage</code>	The string message associated with the most recent exception, if the exception contains a message

#### IMPORTANT

The system assigns the current exception values to `lastEx`, `lastExError`, and `lastExMessage` after the exception message has been displayed on the screen. These variables are not current when you set `BreakOnThrows` to `true` in the Inspector. ▲

### Maintaining View State

---

If you need to maintain the state of a view, store your state information in a soup. If you maintain the view state in a view frame, you lose the state information when the view is closed.

## Debugging

### Accessing the Parent of a View

---

You can access the parent of a view with the `Parent` function. For example:

```
myParent := myView.Parent();
```

Don't use a path name that starts with `_parent`.



# Extended Debugging Functions

---

This document describes the extended NewtonScript debugging functions, which let you study and manipulate an application running on a Newton personal digital assistant.

You can use the functions described in this chapter to

- set break points in an application after it's been compiled and installed
- step through program execution
- examine and change the execution environment
- display a textual representation of the interpreter instructions

**WARNING**

The functions described in this chapter are for debugging purposes only. Do not include them in released products.

## Compatibility

---

You can install the extended debugging functions only on the Newton 2.0 platform.

## Installing the Extended Debugging Functions

---

The extended debugging functions are distributed in a Newton package file named NS Debug Tools.pkg. You download them to the Newton using the Newton Package Installer. To get the most out of the extended debugging functions, install the package named DebugHashToName.pkg as well.

To remove the extended debugging functions, scrub the NS Debug Tools icon in the Extras drawer. To remove DebugHashToName, scrub its icon in the Extensions folder in the Extras drawer.

## Using the Extended Debugging Functions

---

The extended debugging functions let you study NewtonScript functions in a Newton application, through an NTK Inspector window with an open connection to a Newton. The extended debugging functions are executed on the Newton.

When the NTK Compile for Debugging option is set, NTK saves debugging information about each NewtonScript function it compiles. The extended debugging functions use this information—always enable Compile for Debugging when you're compiling code that you plan to examine with the functions described in this chapter.

## Extended Debugging Functions

## Break Loops and Break Points

---

Many of the debugging functions are most useful while the Newton is in a **break loop**—that is, while program execution has been suspended and the Newton is accepting input only from the Inspector connection. When it receives input, the Newton processor evaluates it, prints a visual representation of the value of the last statement evaluated, and resumes waiting for input. This process is the read-evaluate-print (REP) loop.

You can set the `BreakOnThrows` variable to a non-nil value to cause the Newton to go into a break loop whenever an exception is raised.

With the extended debugging functions you can set and manipulate break points in an application that's already compiled and downloaded. While the Newton is in a break loop, you can step through the application and examine and change the state of the program.

### Enabling Break Points

---

To use the break points you've added to functions that are already compiled, you must turn on the interpreter code that checks for them by either:

- checking `Enable Breakpoints` in the NS Debug Tools application, or
- executing the `GloballyEnableBreakPoints` function with a non-nil value as the argument.

Checking for break points slows down execution of all NewtonScript code, whether or not it contains break points. Be sure to disable break points when you're not using the extended debugging functions.

When break points are disabled, the extended debugging functions might not be able to supply complete information in some cases:

- Program counter values might be inaccurate in the display upon entering a break loop and in reports by the `StackTrace`, `Where`, and `GetCurrentPC` functions. If an inaccurate program counter value is possible, the display includes a question mark (?) after the value. You can suppress the warning by placing a slot named `NoInaccWarning` with a non-nil value in the `NSDParamFrame`, described in "Adjusting the Debugging Environment" on page 7-10.

## Extended Debugging Functions

- Occasionally, the function name and the number of arguments are not available. In this case, the display includes as much information as possible.
- Non-interpreted functions might not be reported in the stack trace displays.

You can avoid all of these problems by enabling break points whenever you're using the extended debugging functions, whether or not you're using break points.

## Creating, Removing, and Disabling Break Points

---

You can create a break point in an interpreted function with the `InstallBreakPoint` function. You specify both a function object and a program counter value, which represents an offset into the function. To insert a break point at the beginning of a slider's `changedSlider` method, for example, you would execute a statement something like this:

```
point :=
InstallBreakPoint(Debug("mySlider").changedSlider, 0);
```

If the function is in the current call chain, you can reach the function object with the `GetCurrentFunction` function, which returns the function at the specified place in the call chain. To set a break point in the function that was executing just before a break occurred, for example, you would execute this statement:

```
point := InstallBreakPoint(GetCurrentFunction(0), 0);
```

You can remove break points individually with `RemoveBreakPoint`—passing the break point specification frame returned by `InstallBreakPoint`—or you can remove all break points at once with `RemoveAllBreakPoints`.

You can also disable an individual break point with the `EnableBreakPoint` function, and you can name a break point for later identification with `SetBreakPointLabel`.

## Extended Debugging Functions

The stepping functions create temporary break points, which are removed as soon as they're used.

You can enable or disable all break points in an application without removing them with the function `GloballyEnableBreakPoints`.

### Making Break Points Conditional

---

You can make break points conditional by defining a function named `NSDBreakLoopEntry` that evaluates the circumstances and either cancels or authorizes the break.

The `BreakLoop` function—which is executed when the Newton reaches a break point—looks for a global function with the name `NSDBreakLoopEntry` and, if it finds one, executes it. The function is called with three parameters:

- the name of the function in which the break occurred
- the value of that function's program counter
- an array containing the arguments to the function

If the `NSDBreakLoopEntry` function returns a non-`nil` value, the Newton enters a break loop. If `NSDBreakLoopEntry` returns `nil`, the Newton does not enter the break loop.

When the Newton exits the `BreakLoop` function, it looks for a global function with the name `NSDBreakLoopExit` and, if it finds one, executes it. The function is called with a single argument, which reports whether or not the Newton actually entered the break loop. If `NSDBreakLoopEntry` executed, `NSDBreakLoopExit` receives its return value; if no `NSDBreakLoopEntry` was found, `NSDBreakLoopExit` receives the value `true`.

### Entering a Break Loop

---

When the Newton enters a break loop with the extended debugging functions installed, the Inspector displays

- the name of the current function

## Extended Debugging Functions

- the value of the current program counter and a textual representation of the instruction that the program counter is pointing to
- the break level number

The example here illustrates a break caused by a break just before instruction 25 of a `changedSlider` method.

```
mySlider.changedSlider(), 25: Pop
Entering break loop: level 1
```

## NewtonScript Stacks

---

The NewtonScript interpreter keeps its own collection of data structures. NTK presents the data to you as if there were a single **function-call stack**, which contains an **activation record** for each active function. An activation record is identified by its level on the stack; the current function is at level 0. A stack activation record contains

- a program counter that points to the next instruction that's to be executed
- the receiver and implementor, if any
- the function's parameters, temporary variables, and named variables

You can use the `StackTrace` function to display a summary of the information in the function-call stack. For example,

```
mySlider.changedSlider():25
90112.viewFinalChangeScript(50, 83):29
```

Each line in the display represents one activation record. The current function (`changedSlider` in this example) appears first in the display; it is the record at stack level 0. In this example, `viewFinalChangeScript` is at stack level 1. The display also shows the values of the local variables (in parentheses) and the current program counter value (following the colon).

You can use the stack-access functions—described in “Accessing the Stack” beginning on page 7-16—to examine the functions and their context in more detail and to change stack values.

## Extended Debugging Functions

## Paths to Slots

---

You can use the `GetPathToSlot` and `GetPathWhereSet` functions to find where in a frame's parent chain a specified slot exists and where in the chain the slot's value would change.

Suppose, for example, the tutorial application developed in Chapter 3 is open on the Newton screen. To find out where in the view hierarchy the text slot in the `slowFloaterButton` view exists, you could enter this text:

```
GetPathToSlot(debug("showFloaterButton"), 'text');
```

The text slot is defined in the button's proto template, and so the Newton supplies this path expression:

```
#4419829 _proto.text
```

If you were to change the value of the text slot at run time, however, the change would affect only the current instance. The `GetPathWhereSet` function, therefore, returns a path expression to the text slot within the button's view:

```
GetPathWhereSet(debug("showFloaterButton"), 'text');
#4419A4D text
```

## NewtonScript Byte Code

---

The NTK compiler turns the text—or source code—for a NewtonScript function into a **function object**—that is, a frame containing, among other things, the hardware-independent byte code instructions that are interpreted when the function executes.

You can display a byte code listing of a NewtonScript function with the `Disasm` function. This example shows a simple function and its disassembled instructions:

```
func( )
begin
    SetValue(lowerDisplay, 'text, numberstr(viewValue));
```

## Extended Debugging Functions

```

        :changeLower(viewValue, count);
end

Disasm(Debug("mySlider").changedSlider);
0:  FindVar          lowerDisplay
1:  Push             'text
2:  FindVar          viewValue
3:  Push             'NumberStr
4:  Call             1
5:  Push             'SetValue
6:  Call             3
7:  Pop
8:  FindVar          viewValue
9:  FindVar          count
10: PushSelf
11: Push             'changeLower
14: Send             2
15: Return

```

You can disassemble a part of a function with the `DisasmRange` function.

The disassembly listing shows the offsets to instruction boundaries, which you need for the program counter argument you pass to `InstallBreakPoint` and the stepping function `RunUntil`.

If a function is throwing an exception, you can install a break point at a specific instruction, well before the exception is raised, and then use the stepping functions and the stack-access functions to examine the circumstances leading up to the exception.

The disassembly listing is not identical to the actual byte code—it's merely a textual reconstruction of the binary instruction object. "Interpreter Instructions" beginning on page 7-23 describes the disassembler output.



## Extended Debugging Functions Reference

---

This section describes the extended NewtonScript debugging functions.

### WARNING

The functions described in this chapter are for debugging purposes only. Do not include them in released products. ▲

You can use the extended debugging functions to

- manipulate break points (`InstallBreakPoint`, `RemoveBreakPoint`, `RemoveAllBreakPoints`, `GetAllBreakPoints`, and `GloballyEnableBreakPoints`, plus the optional user-defined functions `NSDBreakLoopEntry` and `NSDBreakLoopExit`)
- step through application execution (`Step`, `StepIn`, `StepOut`, and `RunUntil`)
- access the function-call stack (`StackTrace`, `GetCurrentFunction`, `GetCurrentPC`, `SetCurrentPC`, `Where`, `GetAllTempVars`, `GetTempVar`, `SetTempVar`, `GetAllNamedVars`, `GetNamedVar`, `SetNamedVar`, `GetCurrentReceiver`, and `GetCurrentImplementor`)
- retrieve slot paths (`GetPathToSlot` and `GetPathWhereSet`)
- display a disassembly listing of compiled code (`Disasm` and `DisasmRange`).

## Extended Debugging Functions

## Adjusting the Debugging Environment

---

You can adjust the behavior of some extended debugging functions by defining a global frame named `NSDParamFrame`, in which you place slots with predefined meanings:

<code>verbose</code>	A flag that controls how much information some functions display.
<code>nil</code>	a brief description
<code>non-nil</code>	a longer description
<code>NoInaccWarning</code>	A flag that controls whether or not some functions display a warning when the information returned might be inaccurate. These warnings are necessary only when checking for break points is disabled.
<code>nil</code>	a warning is displayed
<code>non-nil</code>	no warning is displayed

This statement entered in the Inspector window, for example, suppresses the longer comments and the warnings when information returned might be inaccurate.

```
NSDParamFrame := {verbose: nil,
                  noInaccWarning: true
                  };
```

You can also include `NSDParamFrame` slots that specifically control the display of disassembly listings, as described in “Interpreter Instructions” beginning on page 7-23.

## Manipulating Break Points

---

You use the functions described in this section to manipulate break points in an application that’s already compiled and installed on the Newton.

## Extended Debugging Functions

**InstallBreakPoint**

---

`InstallBreakPoint (function, PC)`

Sets a break point within a specified function just before a specified instruction.

*function*                      A function object.

*PC*                              A program-counter value within the function; this value must point to the beginning of a byte code instruction.

The `InstallBreakPoint` function returns a break point specification frame, which you can later pass to `RemoveBreakPoint`.

**RemoveBreakPoint**

---

`RemoveBreakPoint (breakPointSpec)`

Removes the specified break point.

*breakPointSpec*              A break point specification frame, returned previously by `InstallBreakPoint`.

**EnableBreakPoint**

---

`EnableBreakPoint (breakPointSpec, enableMode)`

Enables or disables an individual break point, depending on the value of the *enableMode* parameter.

*breakPointSpec*              A break point specification frame, returned previously by `InstallBreakPoint`.

*enableMode*                      The instruction to turn the break point on or off:

<code>&lt;&gt;nil</code>	Enable the break point.
<code>nil</code>	Disable the break point.

**RemoveAllBreakPoints**

---

`RemoveAllBreakPoints()`

Removes all break points installed by `InstallBreakPoint`.

## Extended Debugging Functions

**GetAllBreakPoints**

---

`GetAllBreakPoints()`

Returns a frame describing all current break points. This frame contains a slot named `programCounter` whose value is an array of break point specifications. Each break point specification contains instructions and `programCounter` slots used by the NewtonScript interpreter to determine when to trigger the break point.

**GloballyEnableBreakPoints**

---

`GloballyEnableBreakPoints(enableMode)`

Enables or disables checking for break points in the NewtonScript interpreter.

<i>enableMode</i>	The instruction to turn checking either on or off:
<code>&lt;&gt;nil</code>	Enable checking for break points.
<code>nil</code>	Disable checking for break points.

The `GloballyEnableBreakPoints` function returns the previous status of breakpoint enabling. It also adjusts the NS Debug Tools package icon in the Extras Drawer to reflect the breakpoint-enable status.

While break points are enabled, the execution of all NewtonScript code—with or without break points—is slowed down while the interpreter checks for break points. Disabling all break points allows NewtonScript code to run at full speed.

You can also enable and disable break points through the Enable Breakpoints checkbox in the NS Debug Tools application on the Newton.

## Extended Debugging Functions

**SetBreakPointLabel**

---

`SetBreakPointLabel (breakPointSpec, label)`

Establishes a label for a specified break point.

<i>breakPointSpec</i>	A break point specification frame, returned previously by <code>InstallBreakPoint</code> .
<i>label</i>	The label, which is placed in the break point specification frame. The label can be a string, a symbol, or any other valid NewtonScript value.

**GetBreakPointLabel**

---

`GetBreakPointLabel (breakPointSpec)`

Returns the label for the specified break point.

<i>breakPointSpec</i>	A break point specification frame, returned previously by <code>InstallBreakPoint</code> .
-----------------------	--

**User-Defined Breakpoint Functions**

---

You can make a break loop conditional or execute your own code on the way into the loop by defining a function with the name `NSDBreakLoopEntry`; you can execute code on the way out of a break loop by defining a function with the name `NSDBreakLoopExit`.

When the Newton enters a break loop with the extended debugging functions installed, the `BreakLoop` function looks for a global function with the name `NSDBreakLoopEntry`. If it finds one, it executes it.

As the Newton exits the `BreakLoop` function, it looks for a global function with the name `NSDBreakLoopExit`. If it finds one, it executes it before resuming program execution.

## Extended Debugging Functions

**NSDBreakLoopEntry**

---

`NSDBreakLoopEntry(function, PC, paramArray)`

Screens a break point or executes user-defined code while entering a break loop.

<i>function</i>	A function object.
<i>PC</i>	A program-counter value within the function. This argument is <code>nil</code> for non-interpreted functions, such as functions in ROM, functions compiled into native code, or C functions.
<i>paramArray</i>	<p>An array containing the arguments to the current function, in one of two formats.</p> <p>If the class of the array is the symbol <code>'paramNameAndValues</code>, the array is organized in paired entries, that is:</p> <pre>[param1Name, param1Value, param2Name,   param2Value, ...]</pre> <p>If the class of the array is the symbol <code>'paramValues</code>, the array contains a series of values, that is:</p> <pre>[param1Value, param2Value, ...]</pre>

You can define an `NSDBreakLoopEntry` function to suppress breaks that do not fit the profile you're interested in. You can check not only the function's name and program counter value but also the values of its arguments.

If your `NSDBreakLoopEntry` function returns `nil`, the Newton does not enter the break loop but instead looks for a function named `NSDBreakLoopExit`.

If your `NSDBreakLoopEntry` function returns any non-`nil` value, the Newton enters a break loop.

## Extended Debugging Functions

**NSDBreakLoopExit**

---

`NSDBreakLoopExit ( didBreakLoopHappen )`

Executes user-defined code while exiting a break loop.

*didBreakLoopHappen* The return value of the `NSDBreakLoopEntry` function, or true if no `NSDBreakLoopEntry` function executed.

You can define an `NSDBreakLoopExit` function to restore any data you've changed in the `NSDBreakLoopEntry` function.

**Stepping**

---

You use the functions described in this section to step through code. These functions are meaningful only while the Newton is in a break loop.

**Step**

---

`Step()`

Executes one byte code instruction and then returns to the REP loop.

**StepIn**

---

`StepIn ( )`

If the current instruction is a function call or a message send, enters the new function and then returns to the REP loop without executing the first instruction. If the function call or message send is a non-interpreted function, `StepIn` displays a warning, does not enter the function, and leaves the program counter unchanged.

**StepOut**

---

`StepOut ( )`

Continues execution until the current function returns, and then returns to the REP loop. If the caller is not an interpreted function, execution stops just before the first instruction of the next NewtonScript function on the function-call stack.

## Extended Debugging Functions

**RunUntil**

---

`RunUntil(function, PC)`

Continues execution until the interpreter reaches the specified location within the specified function.

*function*                      A function object.

*PC*                              A program-counter value within the function; this value must point to the beginning of a byte code instruction.

The `RunUntil` function sets a temporary break point and then allows the Newton to resume execution. When execution reaches the break point, the break point is removed and the Newton returns to the REP loop without executing the next instruction.

**Accessing the Stack**

---

You use the functions described in this section to examine and manipulate the function-call stack. The function that was executing when the Newton entered the break loop is at level 0.

**StackTrace**

---

`StackTrace()`

Displays the names of the functions on the function-call stack, their program counter values, and the values of their parameters.

When the extended debugging functions are installed, the stack trace display—triggered by entering the `StackTrace` function or clicking the Stack Trace button—is as described in this chapter on page 7-6, not as described in Chapter 6, “Debugging.” To display a stack trace in the original format, you can use the `StackTraceOld` function.



## Extended Debugging Functions

**Note**

The debug slots of many system prototypes are encoded as integers, which appear in the stack trace display. If you install the DebugHashToName application on the Newton, the debugging functions can map the integers to strings and produce more readable output. ♦

**GetCurrentFunction**

---

`GetCurrentFunction(level)`

Returns the function object that is currently executing at the specified level of the function-call stack.

*level*                      An integer that specifies which stack frame to examine.

**GetCurrentPC**

---

`GetCurrentPC(level)`

Returns the value of the program counter in the stack frame at the specified level of the stack.

*level*                      An integer that specifies which stack frame to examine.

**SetCurrentPC**

---

`SetCurrentPC(newPC)`

Sets the value of the program counter in the current stack frame. When you continue execution (using `ExitBreakLoop` or any of the stepping functions), execution starts at *newPC*.

*newPC*                      A program counter value.

**Warning**

`SetCurrentPC` is a dangerous function. When you change a program counter, be sure to adjust the values of the temporary and named variables as necessary. Do not change the program counter so that it points to an instruction that needs a different number of temporary variables. ♦

## Extended Debugging Functions

**Where**

---

`Where ( )`

Identifies the current function and displays the value of the current program counter and a textual representation of the byte code instruction the program counter is pointing to.

**GetAllTempVars**

---

`GetAllTempVars ( level )`

Returns an array containing the values of the temporary variables pushed and popped while a function executes.

*level*                      An integer that specifies which stack frame to examine.

**GetTempVar**

---

`GetTempVar ( level, offset )`

Returns the value of the temporary variable at the specified offset into the temporary variable list at the specified level of the stack.

*level*                      An integer that specifies which stack frame to examine.

*offset*                    An integer that specifies which temporary variable to examine. The variable added most recently is at offset 0, the one before that is at offset 1, and so on.

**SetTempVar**

---

`SetTempVar ( level, offset, newValue )`

Sets the value of the temporary variable at the specified offset into the stack of temporary variables at a specified level of the stack.

*level*                      An integer that specifies which stack frame to manipulate.

*offset*                    An integer that specifies which temporary variable to change. The most recent entry is at offset 0.

*newValue*                The new value for the temporary variable.

## Extended Debugging Functions

A temporary variable must already exist at the specified offset; its value is replaced.

**GetAllNamedVars**

---

`GetAllNamedVars ( level )`

Returns a frame containing the names and values of all declared, closed-over, and otherwise locally defined variables at the specified level of the stack.

*level*                      An integer that specifies which stack frame to examine.

The `GetAllNamedVars` function displays a warning when it detects an undeclared local variable, which is technically possible but not efficient programming in NewtonScript.

**GetNamedVar**

---

`GetNamedVar ( level, varNameSymbol )`

Returns the value of the specified variable from the list of named variables at the specified level of the stack.

*level*                      An integer that specifies which stack frame to examine.

*varNameSymbol*          The symbol for the named variable to be examined.

**SetNamedVar**

---

`SetNamedVar ( level, varNameSymbol, newValue )`

Sets the value of the named variable at the in the list of named variables at a specified level of the stack.

*level*                      An integer that specifies which stack frame to examine.

*varNameSymbol*          The symbol for the named variable to be changed.

*newValue*                  The new value of the named variable.

A variable with the specified name must already exist; its value is replaced.

## Extended Debugging Functions

**GetCurrentReceiver**

---

`GetCurrentReceiver(level)`

Returns the current receiver at the specified level of the stack.

<i>level</i>	An integer that specifies which stack frame to examine. The function that was executing when the Newton entered the break loop is at level 0.
--------------	--

The `GetCurrentReceiver` function returns `self`.

The current receiver is the value of `self`.

**GetCurrentImplementor**

---

`GetCurrentImplementor(level)`

Returns the current implementor at the specified level of the stack.

<i>level</i>	An integer that specifies which stack frame to examine.
--------------	---

**Retrieving Paths**

---

This section documents the functions that return path expressions to slots. You can use these functions to search for a slot in a frame, using both parent and proto inheritance rules.

**GetPathToSlot**

---

`GetPathToSlot(aFrame, aSymbol)`

Returns the path expression from the specified frame to the slot with the specified symbol.

<i>aFrame</i>	The frame where the search begins.
---------------	------------------------------------

<i>aSymbol</i>	The symbol for the slot.
----------------	--------------------------

The `GetPathToSlot` function returns the path to the slot whose value would be returned by `GetVariable(aFrame, aSymbol)`.

If the symbol is not found, `GetPathToSlot` returns an empty path expression.

## Extended Debugging Functions

**GetPathWhereSet**

---

*GetPathWhereSet* (*aFrame*, *aSymbol*)

Returns the path expression for the path to the frame in which a specified slot's value would be set if the value of the slot changed.

*aFrame*                      The frame where the search begins.

*aSymbol*                    The symbol for the slot.

The *GetPathWhereSet* function returns the path to the slot whose value would be changed by *SetVariable(aFrame, aSymbol, aValue)*. The function begins its search for the slot in the specified frame and makes use of full proto and parent inheritance.

**Disassembling**

---

You can use the *Disasm* and *DisasmRange* functions to disassemble a block of code or part of a block of code. The disassembly functions produce a textual representation of the function's byte code.

"Interpreter Instructions" beginning on page 7-23 describes the output of the disassembly functions.

**Disasm**

---

*Disasm* (*function*)

Displays a disassembly listing of *function*.

*function*                    A function object.

## Extended Debugging Functions

**DisasmRange**

---

`DisasmRange(function, start, end)`

Displays a disassembler listing of *function* between *start* and *end*.

<i>function</i>	A function object.
<i>start</i>	A program counter value within <i>function</i> where disassembly is to begin.
<i>end</i>	A program counter value within <i>function</i> where disassembly is to end.

**Summary of Extended Debugging Functions**

---

This section summarizes the extended debugging functions.

**Manipulating Break Points**

---

```

InstallBreakPoint(function, PC)
RemoveBreakPoint(breakPointSpec)
EnableBreakPoint(breakPointSpec, enableMode)
RemoveAllBreakPoints()
GetAllBreakPoints()
GloballyEnableBreakPoints(enableMode)
SetBreakPointLabel(breakPointSpec, label)
GetBreakPointLabel(breakPointSpec)
BreakLoop()
NSDBreakLoopEntry(function, PC, paramArray)
NSDBreakLoopExit(didBreakLoopHappen)

```

## Extended Debugging Functions

## Stepping

---

```
Step()  
StepIn()  
StepOut()  
RunUntil(function, PC)
```

## Accessing the Stack

---

```
QuickStackTrace()  
GetCurrentFunction(level)  
GetCurrentPC(level)  
SetCurrentPC(newPC)  
Where()  
GetAllTempVars()  
GetTempVar(level, offset)  
SetTempVar(level, offset, newValue)  
GetAllNamedVars(level)  
GetNamedVar(level, varNameSymbol)  
SetNamedVar(level, varNameSymbol, newValue)  
GetCurrentReceiver(level)  
GetCurrentImplementor(level)
```

## Retrieving Paths

---

```
GetPathToSlot(aFrame, aSymbol)  
GetPathWhereSet(aFrame, aSymbol)
```

## Disassembling

---

```
Disasm()  
DisasmRange(function, start, end)
```

# Interpreter Instructions

---

## Extended Debugging Functions

This section describes the interpreter instructions displayed by the `Disasm` and `DisasmRange` functions and by the Inspector when the Newton enters a break loop with the extended debugging functions installed.

The instructions displayed by the debugging tools are not identical to the actual byte code generated by the NTK compiler—they are rather a textual representation of the compiled function.

You can control the amount of information displayed in the disassembly listings by creating a global frame named `NSDParamFrame`, in which you define one or more of these slots:

<code>verbose</code>	<p>A flag that controls the display of some instructions. The verbose display includes a brief description of the instruction's parameter and the parameter's offset in the current lexical scope.</p> <p><code>nil</code>            no description and offset displayed</p> <p><code>non-nil</code>        description and offset displayed</p>
<code>disasmInstWidth</code>	<p>An integer that specifies the width, in spaces, of the instruction column in a disassembly listing.</p> <p>A value of <code>true</code> invokes the default column width; a value of <code>nil</code> specifies only a space between columns.</p>
<code>disasmArgWidth</code>	<p>An integer that specifies the width, in spaces, of the comment column in a disassembly listing</p> <p>A value of <code>true</code> invokes the default width; a value of <code>nil</code> specifies only a space between the comment column and the third column, which displays the additional information triggered by the verbose flag.</p>

This statement entered in the Inspector window, for example, suppresses the descriptive comments and sets the column widths to the defaults.

```
NSDParamFrame := {verbose:nil,
                  disasmInstWidth:true,
                  DisasmArgWidth:true
                  };
```



## Extended Debugging Functions

## Stack Operations

---

The instructions described in this section manipulate the NewtonScript stack. When a function is called, the parameters are passed on the stack and the result is returned on the stack.

For the most part, if an interpreter instruction uses a value from the stack, it's popped. If there is a result, it's pushed.

### Pop

---

Pop

Pops the top element from the stack.

```
disasm(func() begin Sleep(10); true end);
0: PushConstant      10
3: Push              'Sleep
4: Call              1
5: Pop
6: PushConstant      TRUE
9: Return
```

In this example, the `Pop` instruction appears where it does because the result of the function call isn't used and its value must be removed from the stack.

### PushSelf

---

PushSelf

Pushes the current receiver onto the interpreter stack.

```
disasm(func() self)
0: PushSelf
1: Return
```

## Extended Debugging Functions

**Push**

---

**Push** *X*Pushes the value *X* on the top of the stack.

```
disasm(func() "foo")
  0: Push                "foo"
  1: Return
```

**PushConstant**

---

**PushConstant** *X*Pushes the value *X* on the top of the stack. **PushConstant** is used when *X* is an immediate that fits into 16 bits.

```
disasm(func() nil);
  0: PushConstant        NIL
  1: Return
```

**FindVar**

---

**FindVar** *X*

Looks for the variable named *X* in the lexical context and then in the current receiver (**self**), including both proto and parent inheritance. If that search fails, **FindVar** searches the global variables for *X*. When it finds *X*, **FindVar** pushes its value on the stack. If *X* is not found, **FindVar** pushes **nil** on the stack.

This instruction implements variable lookup as described in *The NewtonScript Programming Language*.

```
disasm(func() x)
  0: FindVar              x
  1: Return
```

## Extended Debugging Functions

You can control the amount of information displayed with the `FindVar` instruction by setting the value of the `verbose` slot in the `NSDParamFrame` frame, as described on page 7-24.

**GetVar**

---

`GetVar` *X*

Gets the parameter or local named *X* and pushes it onto the stack.

```
disasm(func(x) x)
  0: GetVar                x
  1: Return
```

You can control the amount of information displayed with the `GetVar` instruction by setting the value of the `verbose` slot in the `NSDParamFrame` frame, as described on page 7-24.

**MakeFrame**

---

`MakeFrame` *N*

Using the frame map found on the top of the stack, constructs a frame using the next *N* elements of the stack (in bottom-up order) to populate the slots in the frame. The resulting frame is pushed on the stack.

```
disasm(func() {x: 1, y: 2})
  0: PushConstant          1
  1: PushConstant          2
  4: Push                  [#4415A35]
  5: MakeFrame             2
  6: Return
```

## Extended Debugging Functions

**MakeArray**

---

**MakeArray** *N*

Constructs an array of the class of the top element of the stack, placing the next *N* elements of the stack (in bottom-up order) in the array. The resulting array is pushed on the stack.

The default class of an array is `Array`. The instruction `Push Array` may therefore appear just before `MakeArray` for unclassed arrays.

The instruction `MakeArray -1` pops an integer and array class from the stack and allocates an array of that class and length. This is used in the `foreach/collect` statement.

```
disasm(func() [foo: 1, 2])
  0: PushConstant      1
  1: PushConstant      2
  4: Push              'foo
  5: MakeArray         2
  6: Return
```

**GetPath**

---

**GetPath** *N*

Looks for the slot on the top of the stack in the frame that's second on the stack, using `_proto` inheritance only. If *N* is 0 then `nil.y` is `nil`. If *N* is 1 then `nil.y` throws an exception.

```
disasm(func() x.y.z)
  0: GetVar            x
  1: Push              y.z
  2: GetPath          1
  3: Return
```

## Extended Debugging Functions

**SetPath**

---

`SetPath N`

Assigns the value that's on the top of the stack to the slot that's second on the stack in the frame that's third on the stack. If *N* is 0 the result is not put on the stack. If *N* is 1, the result is put on the stack.

```
disasm(func() x.y.z := 1)
  0: FindVar           x
  1: Push              y.z
  2: PushConstant     1
  3: SetPath           1
  4: Return
```

**SetVar**

---

`SetVar X`

Assigns the value on the top of the stack to the local variable or parameter named *X*.

```
disasm(func(x) x := 1)
  0: PushConstant     1
  1: SetVar            x
  2: GetVar            x
  3: Return
```

You can control the amount of information displayed with the `SetVar` instruction by setting the value of the `verbose` slot in the `NSDParamFrame` frame, as described on page 7-24.

**SetFindVar**

---

`SetFindVar X`

Assigns the value on the top of the stack to the variable that's second on the stack, using `FindVar` to locate that variable.

## Extended Debugging Functions

This instruction implements `x := expr`, where `x` is not declared as a local variable.

```
disasm(func() x := 1)
  0: PushConstant      1
  1: SetFindVar         x
  2: FindVar           x
  3: Return
```

You can control the amount of information displayed with the `SetFindVar` instruction by setting the value of the `verbose` slot in the `NSDParamFrame` frame, as described on page 7-24.

### SetLexScope

---

`SetLexScope`

Sets the lexical scope (inherited locals and parameters) of the object on the top of the stack to that of the currently executing function.

```
disasm(func(x) func(y) x+y)
  0: Push              {#4413361}
  1: SetLexScope
  2: Return
```

## Program Flow

---

This section describes the instructions that control program flow.

### While and Repeat/Until Loops

---

Three branching operators provide most of the general program-flow operations, including the implementation of `while` and `repeat/until` loops.

## Extended Debugging Functions

**Branch**

---

*Branch I*

Causes the interpreter to continue processing at the instruction at offset *I*.

```

disasm(func() if x then true)
  0: FindVar           x
  1: BranchIfNil       10
  4: PushConstant      TRUE
  7: Branch            11
 10: PushConstant      NIL
 11: Return

```

**BranchT**

---

*BranchT I*

If the top of the stack is non-nil, causes the interpreter to continue processing at the instruction at offset *I*.

```

disasm(func() while x do y)
  0: Branch            5
  3: FindVar           y
  4: Pop
  5: FindVar           x
  6: BranchIfNotNil    3
  7: PushConstant      NIL
  8: Return

```

**BranchF**

---

*BranchF I*

If the top of the stack is nil, causes the interpreter to continue processing at the instruction at offset *I*.

```

disasm(func() if x then true)
  0: FindVar           x

```

## Extended Debugging Functions

```

1: BranchIfNil      10
4: PushConstant    TRUE
7: Branch          11
10: PushConstant   NIL
11: Return

```

## For Loops

---

The `for` loop implementation uses a triplet of integers (a loop counter, a limit value, and a counter increment) as loop variables. The byte code instructions increment the loop counter and determine if the loop is done.

The current value of the loop counter is identified by the name of the variable; the increment and end values are identified by the suffixes `|incr` and `|limit` on the variable name. If the loop counter is `i`, for example, then the other loop variables are `i|incr` and `i|limit`.

A simple branch exits the loop. Because the iterator counter and limit are pseudo-local variables, nothing special needs to be done to clean them up on breaks.

## IncrVar

---

`IncrVar X`

Increments the local variable or parameter named `X` by the amount at the top of the stack and pushes the result onto the stack. This instruction doesn't pop the increment value, which remains on the stack.

```
disasm(func() for x := a to b by c do getappparams())
```

```

0: FindVar      a
1: SetVar       x
2: FindVar      b
3: SetVar       x|limit
4: FindVar      c
5: SetVar       x|incr
6: GetVar       x|incr

```



## Extended Debugging Functions

```

7: GetVar          x
8: Branch          16
11: Push           'GetAppParams
12: Call           0
13: Pop
14: GetVar          x|incr
15: IncrVar         x
16: GetVar          x|limit
17: BranchIfLoopNotDone 11
20: PushConstant   NIL
21: Return

```

**BranchIfLoopNotDone**


---

BranchIfLoopNotDone *I*

Determines if the loop is complete, using the top three elements of the stack as the current loop variable, the loop limit, and the loop increment. If the loop is not complete, this instruction branches to *I*.

```
disasm(func() for x:= 0 to 9 do nil)
```

```

0: PushConstant    0
1: SetVar           x
2: PushConstant    9
5: SetVar           x|limit
6: PushConstant    1
7: SetVar           x|incr
8: GetVar           x|incr
9: GetVar           x
10: Branch          15
13: GetVar          x|incr
14: IncrVar         x
15: GetVar          x|limit
16: BranchIfLoopNotDone 13

```

## Extended Debugging Functions

```

19: PushConstant      NIL
20: Return

```

## Foreach Loops (Frame and Array Iterators)

The implementation of the foreach loop uses an “iterator” data structure that tracks progress through the array. A frequently used function (number 17) is used to create a new iterator for the object, and the instructions `IterNext` and `IterDone` increment and test this iterator.

**IterNext**

`IterNext`

Increments the array or frame iterator.

```

disasm(func() foreach s, v in x do getappparams())
  0: FindVar          x
  1: PushConstant     NIL
  2: NewIterator      2
  5: SetVar           sv|iter
  6: Branch           22
  9: GetVar           sv|iter
 10: PushConstant     1
 11: Aref             2
 12: SetVar           v
 13: GetVar           sv|iter
 14: PushConstant     0
 15: Aref             2
 16: SetVar           s
 17: Push            'GetAppParams
 18: Call             0
 19: Pop
 20: GetVar           sv|iter
 21: IterNext
 22: GetVar           sv|iter

```

## Extended Debugging Functions

```

23: IterDone
24: BranchIfNil          9
27: PushConstant        NIL
28: PushConstant        NIL
29: SetVar               sv|iter
30: Return

```

**IterDone**

---

IterDone

Tests whether an array or frame iterator is complete and pushes result of the test on the stack.

```

disasm(func() foreach elt in x do nil)
 0: FindVar              x
 1: PushConstant         NIL
 2: NewIterator          2
 5: SetVar               elt|iter
 6: Branch               15
 9: GetVar               elt|iter
10: PushConstant         1
11: Aref                 2
12: SetVar               elt
13: GetVar               elt|iter
14: IterNext
15: GetVar               elt|iter
16: IterDone
17: BranchIfNil          9
20: PushConstant        NIL
21: PushConstant        NIL
22: SetVar               elt|iter
23: Return

```

## Extended Debugging Functions

## Exception Handling

---

The interpreter maintains a stack of exception-handler contexts, each of which represents the dynamic scope of a `try/onexception` statement for a single function object.

A pair of handler instructions registers and removes contexts on the exception-handler stack. When an exception occurs, the interpreter first checks the object's handlers and branches to one of them if appropriate.

### NewHandlers

---

`NewHandlers` *N*

Registers the top *N* pairs of elements on the stack as exception handlers for the currently executing function. Within each pair of items, the second (lowest) is the exception symbol and the first (highest) is the instruction number to jump to to process that exception.

```
disasm(func() try nil onexception |evt.ex.msg| do true;
      onexception |evt.ex| do 'foo )
0: Push                               'evt.ex.msg
1: PushConstant                       16
4: Push                               'evt.ex
5: PushConstant                       22
8: NewHandlers                        2
9: PushConstant                       NIL
10: PopHandlers
13: Branch                           26
16: PushConstant                       TRUE
19: Branch                           23
22: Push                               'foo
23: PopHandlers
26: Return
```

## Extended Debugging Functions

**PopHandlers**

---

## PopHandlers

Removes the most recently added set of exception handlers from the exception-handler stack. (See `NewHandlers`.)

```
disasm(func() try true onexception |evt.ex| do 'foo)
  0: Push                'evt.ex
  1: PushConstant        14
  4: NewHandlers         1
  5: PushConstant        TRUE
  8: PopHandlers
 11: Branch              18
 14: Push                'foo
 15: PopHandlers
 18: Return
```

**Calling and Returning Functions**

---

In NewtonScript, you can invoke the execution of a function in a number of different ways:

- message sends
- conditional sends
- direct calls

To support these different calling strategies, the interpreter uses a number of different instructions, documented in this section. The interpreter invokes the `Perform` and `Apply` functions as it would any other global function, that is, with the `Call` instruction.

A single instruction controls function return.

The interpreter optimizes the calling of a few functions that are expected to be called often. The section “Primitive Functions” beginning on page 7-40 describes the optimized functions.

## Extended Debugging Functions

**Call**

---

`Call N`

Executes the global function whose symbol is on top of the stack, passing *N* elements on the stack as arguments.

```
disasm(func() GetAppParams())
  0: Push                'GetAppParams
  1: Call                0
  2: Return
```

**Invoke**

---

`Invoke N`

Executes the function on the top of the stack, using its closed-over message context and passing the next *N* stack elements as arguments.

```
disasm(func(x) call x with (y))
  0: FindVar              y
  1: GetVar               x
  2: Invoke              1
  3: Return
```

This instruction is the same as `Call`, except that `Call` finds the name of the global function on the top of the stack, and `Invoke` finds the function itself on the top of the stack.

**Send**

---

`Send N`

Sends the message on the top of the stack to the object that's second on the stack, passing the next *N* elements on the stack as arguments.

```
disasm(func(x, y) x:msg(y))
  0: GetVar              y
  1: GetVar              x
```

## Extended Debugging Functions

```

2: Push                'msg
3: Send                1
4: Return

```

**SendIfDefined**

---

SendIfDefined *N*

Attempts to send the message on the top of the stack to the object that's second on the stack, passing the next *N* elements on the stack as arguments. If a full proto/parent lookup does not find the specified message in the object, SendIfDefined pushes nil on the stack. This instruction is like send, but it implements the :? syntax.

```

disasm(func(x, y) x:?msg(y))
0: GetVar              y
1: GetVar              x
2: Push                'msg
3: SendIfDefined       1
4: Return

```

**Resend**

---

Resend *N*

Sends the message on the top of the stack using the current message context, passing the next *N* elements on the stack as arguments. The Resend instruction starts searching for the method to invoke in the proto slot, if any, of the current implementor. This instruction implements the inherited: syntax.

```

disasm(func() inherited:msg())
0: Push                'msg
1: Resend              0
2: Return

```

## Extended Debugging Functions

**ResendIfDefined**

---

**ResendIfDefined** *N*

Attempts to send the message on the top of the stack using the current message context, passing the next *N* elements on the stack as argument. The **ResendIfDefined** instruction starts looking for the method to invoke in the proto slot, if any, of the current implementor. This instruction implements the `inherited:?` syntax.

**Return**

---

**Return**

Returns from the function. The result of the function remains on the top of the stack.

```
disasm(func() nil);
  0: PushConstant      NIL
  1: Return
```

**Primitive Functions**

---

Some NewtonScript operations are not implemented directly as byte code instructions but are defined as **primitive functions**—that is, operations that are performed like function calls.

The primitive functions include

- elements of the NewtonScript language, documented in *The NewtonScript Programming Language*
- functions used by the interpreter itself

The rest of this section lists the primitive functions, in this form:



## Extended Debugging Functions

*Name NumberOfStackElements**Description**Example***Add**

---

+            2

Adds together the first two elements on the stack.

disasm(func() x+y)

0: FindVar	x
1: FindVar	y
2: +	2
3: Return	

**Subtract**

---

-            2

Subtracts the top element on the stack from the second element on the stack and pushes the result.

disasm(func() x-y)

0: FindVar	x
1: FindVar	y
2: -	2
3: Return	

**Multiply**

---

\*            2

Multies the first two elements on the stack.

disasm(func() x \* y)

0: FindVar	x
1: FindVar	y

## Extended Debugging Functions

```

2: *                2
5: Return

```

**Divide**

---

```

/                2

```

Divides the second element on the stack by the first.

```

disasm(func() x / y)
0: FindVar          x
1: FindVar          y
2: /                2
5: Return

```

**Div**

---

```

Div            2

```

Divides the second element on the stack by the first and truncates the remainder to a whole number.

```

disasm(func() x div y)
0: FindVar          x
1: FindVar          y
2: Div              2
5: Return

```

**ARef**

---

```

ARef            2

```

Dereferences an array or string, using the stack elements this way:  
*topOfStack[secondOnStack]*

```

disasm(func() x[y])
0: FindVar          x
1: FindVar          y

```

## Extended Debugging Functions

```

2: ARef          2
3: Return

```

**SetARef**

---

```
SetARef          3
```

Assigns a string or array.

```

disasm(func() x[y] := z)
0: FindVar          x
1: FindVar          y
2: FindVar          z
3: SetARef          3
4: Return

```

**NewIterator**

---

```
NewIterator      2
```

Creates an iterator data structure for an object. This function supports foreach loops.

```

disasm(func() foreach elt in x do nil)
0: FindVar          x
1: PushConstant     NIL
2: NewIterator      2
5: SetVar           elt|iter
6: Branch           15
9: GetVar           elt|iter
10: PushConstant    1
11: Aref            2
12: SetVar           elt
13: GetVar           elt|iter
14: IterNext
15: GetVar           elt|iter
16: IterDone

```

## Extended Debugging Functions

```

17: BranchIfNil          9
20: PushConstant        NIL
21: PushConstant        NIL
22: SetVar               elt|iter
23: Return

```

**Length**

---

```
Length          1
```

Returns the number of elements in the array on the top of the stack.

```

disasm(func() length(x))
  0: FindVar          x
  1: Length           1
  4: Return

```

**AddArraySlot**

---

```
AddArraySlot   2
```

Appends a new element onto an array.

```

disasm(func() AddArraySlot(x, y))
  0: FindVar          x
  1: FindVar          y
  2: AddArraySlot     2
  5: Return

```

**Equals**

---

```
=          2
```

Tests the top two elements on the stack for equality.

```

disasm(func() x = y)
  0: FindVar          x
  1: FindVar          y

```

## Extended Debugging Functions

```

2: =                2
3: Return

```

**NotEquals**

---

```
<>                2
```

Tests the first two elements on the stack for inequality.

```

disasm(func() x <> y)
0: FindVar                x
1: FindVar                y
2: <>                    2
3: Return

```

**LessThan**

---

```
<                2
```

Compares for inequality: Is the second element on the stack less than the first?

```

disasm(func() x < y)
0: FindVar                x
1: FindVar                y
2: <                    2
5: Return

```

**GreaterThan**

---

```
>                2
```

Compares for inequality: Is the second element on the stack greater than the first?

```

disasm(func() x > y)
0: FindVar                x
1: FindVar                y

```

## Extended Debugging Functions

```

2: >                2
5: Return

```

**GreaterOrEqual**

---

```
>=                2
```

Compares for inequality: Is the second element on the stack greater than or equal to the first?

```

disasm(func() x >= y)
0: FindVar          x
1: FindVar          y
2: >=                2
5: Return

```

**LessOrEqual**

---

```
<=                2
```

Compares for inequality: Is the second element on the stack less than or equal to the first?

```

disasm(func() x <= y)
0: FindVar          x
1: FindVar          y
2: <=                2
5: Return

```

**Not**

---

```
Not                1
```

Tests the top element on the stack for nil.

```

disasm(func() not x)
0: FindVar          x

```

## Extended Debugging Functions

```

1: Not          1
2: Return

```

**BitAnd**

---

```

BAnd          2

```

Performs a binary and on the first two elements of the stack.

```

disasm(func() band(x, y))
0: FindVar          x
1: FindVar          y
2: BAnd             2
5: Return

```

**BitOr**

---

```

BOr           2

```

Performs a binary or on the first two elements on the stack.

```

disasm(func() bor(x, y))
0: FindVar          x
1: FindVar          y
2: BOr              2
5: Return

```

**BitNot**

---

```

BNot          2

```

Performs a binary not on the first two elements on the stack.

```

disasm(func() bnot(x, y))
0: FindVar          x
1: FindVar          y
2: BNot             2
5: Return

```

## Extended Debugging Functions

**Clone**

---

Clone 1

Makes a “shallow copy” of the object on the top of the stack.

```
disasm(func() clone(x))
  0: FindVar          x
  1: Clone            1
  4: Return
```

**SetClass**

---

SetClass 2

Sets the class of the object.

```
disasm(func() SetClass(x, y))
  0: FindVar          x
  1: FindVar          y
  2: SetClass         2
  5: Return
```

**Stringer**

---

Stringer 1

Concatenates strings, supporting the & keyword.

```
disasm(func() x&y)
  0: FindVar          x
  1: FindVar          y
  2: Push             'Array
  3: MakeArray        2
  4: Stringer         1
  7: Return
```

```
disasm(func() x&&y)
  0: FindVar          x
```



## Extended Debugging Functions

```

1: Push                " "
2: FindVar             y
3: Push                'Array
4: MakeArray           3
5: Stringer            1
8: Return

```

**HasPath**

---

```
HasPath          2
```

Checks for the existence of an object.

```

disasm(func() x.y exists)
0: FindVar           x
1: Push              'y
2: HasPath            2
5: Return

```

**ClassOf**

---

```
ClassOf          1
```

Returns the class of the object on the top of the stack.

```

disasm(func() ClassOf(x))
0: FindVar           x
1: ClassOf            1
4: Return

```

## Extended Debugging Functions

# Tuning Performance

---

You can use NTK to collect performance statistics on your functions and to optimize selected functions for speed, as described in this chapter.

Chapter 6, “Debugging,” describes the functions you use to examine memory use and drawing efficiency.

## Measuring Performance

---

The profiler times selected functions as they execute on the Newton device. The statistics are displayed at your request in the Inspector window on the development system.

The profiler runs on any Newton MessagePad 120 or later model. To use the profiler on an English-language MessagePad 100 or 110, install the appropriate patch using the Newton Package Installer. The patches are shipped with NTK in a directory named System Updates.

To collect performance statistics, you

## Tuning Performance

- mark the functions you want to profile, as described in “Marking Functions for Profiling” beginning on page 8-2
- select Compile for Profiling through the Project tab in the Settings dialog box, as described in “Configuring the Compiler for Profiling” beginning on page 8-4
- build and download the application
- turn on profiling on the Newton in the Toolkit application, as described in “Configuring the Profiler on the Newton” beginning on page 8-6
- run the code to be profiled
- upload the statistics

When you’re done profiling, be sure to turn off compiler profiling through Project in the Settings dialog box. Before shipping your application, verify that the release build does not contain profiling code.

## Marking Functions for Profiling

---

You turn statistics collection on and off during execution by bracketing code you want profiled with calls to the `EnableProfiling` function. You pass a parameter of `true` to turn profiling on, a parameter of `nil` to turn it off. The function returns the previous state of the profiler.

The following example shows a test method that retrieves and sorts an array, in order to time the sorting function `bubbleSort`.

```
func(vector, size)
begin
    if kProfileOn then                                // Compile for Profiling
                                                        // is set
        local pFlag :=                                // save state;
            EnableProfiling(nil);                      // turn profiling off
    :InitArray(vector);                                // don't profile init routine
    if kProfileOn then
        EnableProfiling(true);                        // turn profiling on
    :bubbleSort(vector, size);                        // execute bubbleSort
```

## Tuning Performance

```

if kProfileOn then                                // always check for profiling
    EnableProfiling(pFlag);                       // restore profiler state
end

```

The `EnableProfiling` function is available only when Compile for Profiling is enabled, which you can test by checking the value of the `kProfileOn` constant. By testing the constant before making any calls to `EnableProfiling`, you can leave profiling code in place in your source code. (When the compiler evaluates `if kProfileOn` to `nil`, it strips the statement from its output.)

The profiler records and times all functions that are executed between the time profiling is turned on and the time it's turned off, including functions that are called indirectly. In this example, the profile reflects the execution of the `bubbleSort` method and any other functions it invokes. Each function appears separately in the profile.

Suppose, for example, that `bubbleSort` in the above example is defined as

```

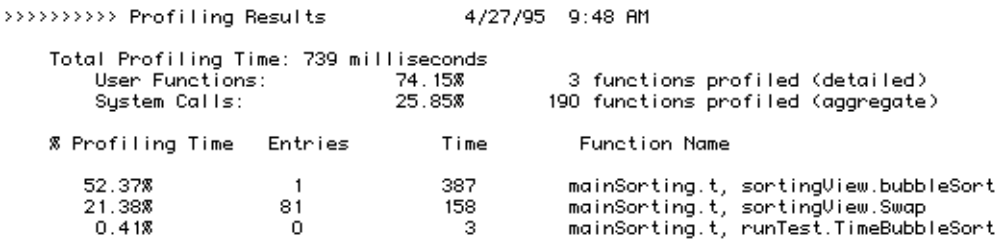
func(vector, size)
begin
    local i, j, element;
    for i := 0 to size-2 do
        begin
            for j := 0 to size-i-2 do
                begin
                    if vector[j+1] < vector[j] then
                        :Swap(vector, j, j+1);
                    end;
                end;
            end;
        end;
    end;
end;

```

The time reported for `bubbleSort` does not include time spent in the `Swap` method, which is reported separately, as illustrated in Figure 7-1.

# Tuning Performance

**Figure 7-1** A performance profile



“Interpreting a Profile” beginning on page 8-8 describes the display.

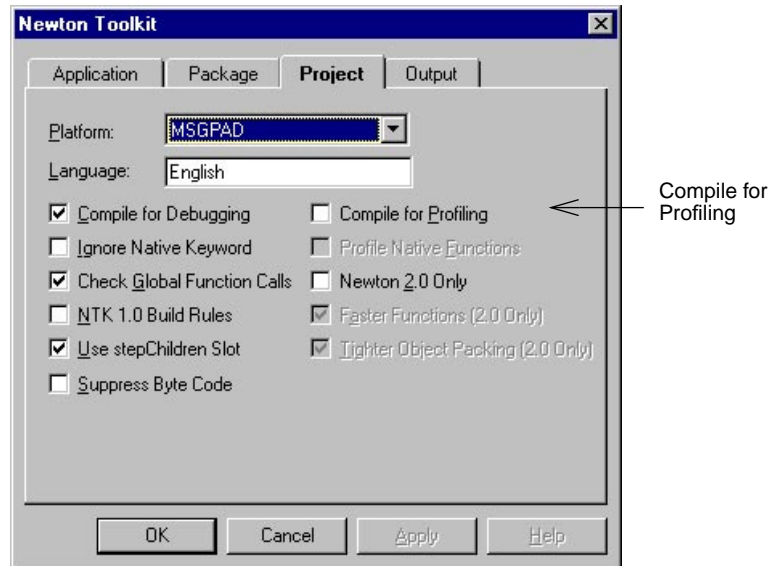
The profiling itself adds a disproportionate amount of execution time to functions compiled into native code. You can reduce the impact of profiling by collapsing the reports of native functions that call other native functions, as described in “Profiling Native Functions” beginning on page 8-19.

NTK keeps one set of statistics on the Newton. You may add to the statistics with paired calls to `EnableProfiling`.

## Configuring the Compiler for Profiling

You instruct the NTK compiler to embed profiling code in its output through the Project Settings dialog box, illustrated in Figure 7-2.

## Tuning Performance

**Figure 7-2** The Project Settings dialog box

Check **Compile for Profiling** to turn profiling on. While **Compile for Profiling** is checked, the compiler assigns each function in the application a unique identifier, which it maps to the source code, and it recognizes the calls that enable and disable profiling. You can test for profiling by checking the `kProfileOn` constant, which is true when **Compile for Profiling** is checked.

The **Profile Native Functions** option instructs the compiler to embed profiling code even in functions that have been compiled into native code. As described in “Profiling Native Functions” beginning on page 8-19, the process of profiling adds significant distortion to native functions. If this option is not checked, NTK compiles native functions as it would in an ordinary build, with the result that the profiler can’t distinguish a native function called from within another native function. A profile generated with this option unchecked shows only the native functions that were called

## Tuning Performance

directly from interpreted functions. The times reported include processor time spent in any other native function calls.

## Configuring the Profiler on the Newton

---

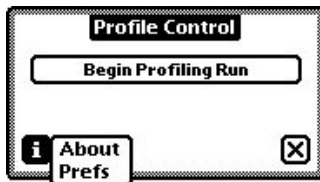
You control profiling on the Newton through the Profile Control view, illustrated in Figure 7-3, which you reach by tapping Profile Control in the Toolkit application.

**Figure 7-3** Profile Control on the Newton



To configure the profiler, choose Prefs from the Info pop-up menu in the lower-left corner of the view, as illustrated in Figure 7-4.

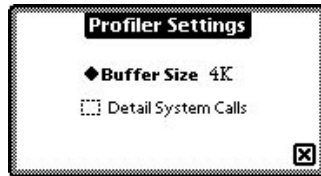
**Figure 7-4** Profiler Info



Choosing Prefs displays the Profiler Settings view, illustrated in Figure 7-5.



## Tuning Performance

**Figure 7-5** Profiler Settings on the Newton

The Buffer Size setting lets you specify the amount of Newton memory devoted to statistics storage. The profiler needs at least 4K of RAM. It's best to keep the allocation as small as possible, to minimize impact on the Newton, but you can increase it if you need to. If the profiler runs out of storage space, it turns itself off and reports that the results are incomplete.

The Detail System Calls checkbox instructs the profiler to track system calls as thoroughly as it tracks your own functions. When this box is not checked, the profiler reports only the total number of system call invocations and the percentage of profiling time they represent. When this box is checked, the profiler tracks and reports system calls individually. It tells you how often each was executed and how much processor time it took.

## Collecting Statistics

When you've compiled an application for profiling and downloaded it, you set up the Newton for profiling through the Toolkit application.

- Open the Toolkit by tapping its icon in the Extras drawer.
- If there is not already an Inspector connection open, open one.
- Tap Profile Control to open the profiler view
- Tap Begin Profiling Run
- Execute the application to be profiled
- Tap Upload Results to display the statistics in the Inspector window.

## Tuning Performance

If you need the screen space, you can close the Toolkit application once you've tapped Begin Profiling Run. When you're ready to upload the statistics, open the Toolkit application again and tap Upload Results.

## Interpreting a Profile

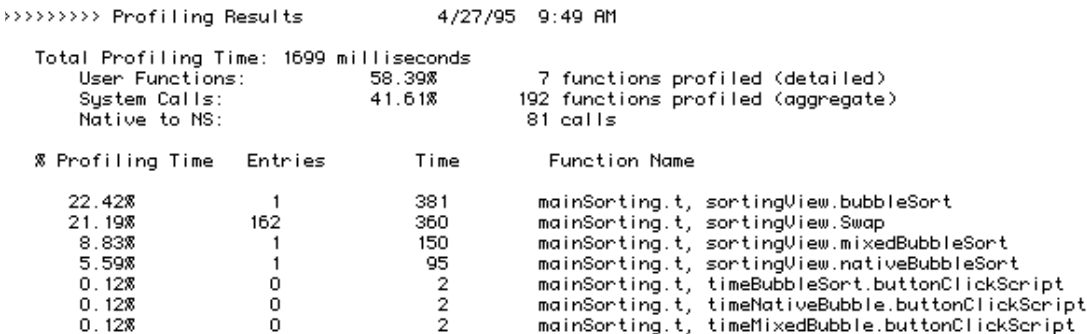
The first part of the profile provides a summary of the profiling:

- the total execution time spent in profiling
- the number of user functions profiled and the percentage of profiling time they represent
- the number of system calls profiled and the percentage of profiling time they represent.

If garbage collection was performed during the profiling run, an entry for it appears in the summary. If system functions created closures during the profile, the total time spent in these closures is reported in an entry labeled Other.

If there were calls from a native function to an interpreted function, those calls are reported on the last line of the summary, as illustrated in Figure 7-6.

**Figure 7-6** A performance profile



## Tuning Performance

The second part of the profile lists the profiled functions in descending order by the amount of processor time each used while profiling was enabled. Figure 7-6 illustrates the profile of three variations on the bubbleSort example used in “Marking Functions for Profiling” beginning on page 8-2.

The Entries column reports the number of times the function was executed. A zero in this column reflects a function that occupied processor time while profiling was in effect but whose entry point was not logged—that is, a function that was started before profiling was enabled but whose execution occupied some processor time during profiling, like the `buttonClickScript` methods for the buttons that triggered the profiling tests in the example in Figure 7-6.

The Time column reports the total processor time, in milliseconds, used by the function.

The profile identifies by name functions defined in the template files. Functions defined in text files (with the `DefineGlobalConstant` function, described in “Defining Global Constants” beginning on page 4-37) appear under the name of the text file, with a number if necessary. If a text file named `projectData` defines a single profiled function, for example, the function appears in the profile with this name:

```
projectData.text
```

If a text file named `projectData` defines two profiled functions, they appear in the profile under these names:

```
projectData.text.1
projectData.text.2
```

The profiler identifies an anonymous nested function by appending an integer to the path name of its parent. Consider, for example, a function in a slot named `myfunc`:

```
func( )
  begin
    local pos := ArrayPos(cardSoups, soupName, 0,
                          func(x,y) ClassOf(y) =
```

## Tuning Performance

```

        'String and StrEqual(x,y));
    . . .
end

```

The fourth argument to `ArrayPos` is an anonymous function, which appears in the profile as `myLayout.myView.myFunc.1`. A second anonymous function within `myFunc` would appear with the name `myLayout.myView.myFunc.2`, and so on.

The reporting of native functions depends on the setting of the Profile Native Functions option in the Project tab in the Settings dialog box, as described in “Profiling Native Functions” beginning on page 8-19.

The profiler depends on tables created during the build to match the functions profiled on the Newton with their names in the source files. Once you’ve shut down NTK, you must rebuild and download the application again before you can match newly collected statistics to the source code.

## Compiling Functions for Speed

---

You can instruct NTK to compile individual functions into native code—that is, the machine language used by the ARM processor. By default, NTK produces machine-independent byte code to be processed by the NewtonScript interpreter.

The native version of a function can execute many time faster than the byte code version, but it is also many times larger. For the most effective balance between speed and size, compile only the most frequently used functions—or the most time-critical—into native code.

You can turn native compiling on or off for a build through the Project Settings item in the Project menu. For compatibility with future platforms, NTK places in the application both the byte code and the native code versions of marked functions. You can reduce the size slightly—at the expense of future incompatibility—by suppressing the byte code through Project Settings.

## Tuning Performance

To mark an individual function for native compiling, construct it with the `native` keyword:

```
func native (paramList) expression
```

The function

```
func (a, b, c) . . .
```

for example, becomes

```
func native (a, b, c) . . .
```

If native-code compiling is turned off through Project Settings, NTK ignores the word `native` in the function statement. The `native` keyword is ignored in the Inspector.

Note that native functions are not locked and are therefore not safely put in the heap. Don't deep clone a native function and then execute it.

A function compiled into native code can have no more than five arguments.

## Declaring and Typing Variables

---

To get the fastest performance in a native function, declare all local variables and specify the types of variables whenever possible.

Native code uses the same mechanism as the interpreter to look up inherited variables and undeclared local variables. Both the native compiler and the interpreter are much more efficient when handling declared local variables. Always declare all local variables, either explicitly with the `local` statement or implicitly with a `for` or `foreach` loop.

You can type your local variables by placing a specifier in the `local` statement, between the keyword `local` and the name of the first variable.

For example:

```
local int x, y := 13;
local array a;
```

All the local variables defined in one statement are of the specified type.

## Tuning Performance

NTK recognizes two type descriptors:

- `int`, which specifies an integer
- `array`, which specifies an array reference.

Local variables you create without a type descriptor are allowed to hold any kind of value.

You can type function arguments by inserting a type before the argument's name in the function statement:

```
func native (int a, b, array c) . . .
```

In this example, the argument `a` is an integer, `b` is untyped, and `c` is an array pointer.

The advantages of typing variables arise when the compiler can optimize an operation using the known types of the operands. Notably, arithmetic operations on `int` expressions are much faster than on untyped expressions.

Specifying a type for a variable restricts the values it can contain. The generated code is type-checked at run time when checking is necessary to ensure that typed variables contain values of the correct type—that is, whenever a result of a broader type is assigned to a variable of a narrower type. For example,

```
local int i := SomeFunction();
```

generates a run-time check to ensure that `SomeFunction` returns an integer. No checking is required, however, in the subsequent assignment

```
local int j := i;
```

because `i` is guaranteed to contain an integer.

The index variables in a `for` loop are automatically declared as integers. If an index variable is also declared with a different type or without a type in a `local` statement, the compiler issues a warning and uses the broader type. For example:

## Tuning Performance

```
func native ()
begin
    local i;
    for i := 1 to 10 do nil;
end
```

The compiler issues a warning and uses an untyped variable to hold the value of `i`.

## Stepping Through an Array

---

A native `for` loop is much faster than a native `foreach` loop for stepping through an array.

## Handling Exceptions

---

The system software maintains separate exception stacks for native and interpreted execution. A native function called from within a non-native function's `onexception` block, therefore, cannot use the `CurrentException` function to access the exception being handled by the non-native function. A native function can call `CurrentException` only from within its own `onexception` block.

*The NewtonScript Programming Language* describes exception handling and the `CurrentException` function.

## Calling Other Functions

---

Byte code is executed by the NewtonScript interpreter; native functions are executed directly by the processor. Because each has its own stacks and registers, transitions between the two execution environments affect performance.

In general, the NewtonScript interpreter can call a native function and then return with little overhead. When a native function is executing, however, a

Tuning Performance

call to an interpreted method invokes the relatively slow process of starting up the interpreter.

The full impact of calls between different function types also depends on how the call is made.

Calling Options

In NewtonScript, functions can be called in three different ways: globally, with the `call/with` syntax, or with a message send.

Global Function Call

You can call a global function directly by name. For example,

```
RefreshViews();
```

When executing a global function call, the system looks up the function by name and then executes it.

NTK provides an optimized dispatch mechanism that bypasses the lookup when you call certain common utility functions from within a native function. These functions' locations are known at compile time, and they are executed directly. The functions cannot therefore be redefined at run time (a practice that is possible but discouraged).

Table 7-1 lists the optimized functions.

**Table 7-1** Utility functions optimized for calling as global functions from a native function

AddArraySlot	DeepClone	IsInstance	StuffByte
ArrayMunger	Downcase	IsString	StuffChar
ArrayPos	EndsWith	IsSubclass	StuffCString
Band	EnsureInternal	IsSymbol	StuffLong
BeginsWith	ExtractByte	Length	StuffPString



## Tuning Performance

BinaryMunger	ExtractBytes	ReplaceObject	StuffUniChar
Bnot	ExtractChar	SetClass	StuffWord
Bor	ExtractCString	SetLength	StuffXLong
Bxor	ExtractLong	Sort	Substr
Capitalize	ExtractPString	StrMunger	TotalClone
CapitalizeWords	ExtractUniChar	StrPos	TrimString
ClassOf	ExtractWord	StrReplace	Uppcase
Clone	ExtractXLong		

**Call/With Syntax**

---

You can call a function with the `call` and with keywords. For example,

```
call myFunction with (x, y);
```

When the system executes a function called with this syntax, it can skip the function lookup and thus complete the call faster.

This syntax saves time as long as the function expression is simple—if, for example, you call the function with a local variable or a constant. You can use the constants supplied in the platform files with the `call/with` syntax, as described in “Platform Files” on page 4-30.

One of the standard NewtonScript optimization strategies is to cache a global function that’s called from inside a loop in a local variable to avoid repeated lookup. For example:

```
local fn := function.SomeGlobalFunction;
for i := 1 to 100000 do call fn with (x, y);
```

**Message Send**

---

You can send a message that causes a function to execute. For example,

## Tuning Performance

```
self:myFunction(x, y);
```

The `send` operation looks up the message in the receiver's inheritance chain, and then performs a variation on a function call. The function execution itself is essentially the same speed as `call/with`, but the lookup is generally more complicated and thus slower.

## Timing Interactions

---

The native NewtonScript compiler normally generates code that tries to eliminate as much overhead as possible. The code determines at run time whether a function being called is native and if it is, bypasses the interpreter's function-call operation.

The price of this optimization is that a call from one native function to another is invisible to both the profiler and the tracing system. You can force these calls through the interpreter—to make them available for profiling and tracing—by checking the Profile Native Functions option in Project Settings, as described in “Configuring the Compiler for Profiling” beginning on page 8-4.

Table 7-2 shows the operations required to do all combinations of function-call operations and function-type transitions. The table uses these operation codes:

- II—the interpreter's calling an interpreted function
- IN—the interpreter's calling a native function
- NN—a direct native function call
- O—an optimized native function call, that is, a call from a native function to one of the optimized functions listed in Table 7-1
- GL—a global-function lookup
- ML—a message lookup
- SI—starting up the interpreter

Calls from native functions to native functions are fastest; the operations have these relative speeds:

Tuning Performance

$O < NN < IN < II$

Thus, it's quicker to call even a non-optimized native function from another native function than to call a native function from an interpreted function.

**Table 7-2**      Function call operations

	<b>Calling an interpreted function from an interpreted function</b>	<b>Calling a native function from an interpreted function</b>	<b>Calling an interpreted function from a native function</b>	<b>Calling a native function from a native function</b>
<b>Call/with</b>	II	IN	SI + II	NN
<b>Global</b>	GL + II	GL + IN	GL + SI+ II	GL + NN
<b>Optimized global</b>	always native	GL + IN	always native	O
<b>Message send</b>	ML + II	ML + IN	ML + SI+ II	ML + NN

Figure 7-6 on page 8-8 illustrates the combined profile of a function executed in three variations:

- an interpreted function (bubbleSort) that calls another interpreted function (Swap)
- a native function (nativeBubbleSort) that calls another native function (which doesn't show up in the profile)
- a native function (mixedBubbleSort) that calls an interpreted function (Swap).

## An Optimization Example

Suppose you've found through profiling that you're spending a lot of time in this binary search function (which searches array a for entry k):

```
func (a, k)
  begin
```

## Tuning Performance

```

local low := 0, high := Length(a)-1, mid;
while high >= low do
  begin
    mid := (low + high) div 2;
    if a[mid] > k then high := mid - 1
      else if a[mid] < k then low := mid + 1
        else return mid;
    end;
  nil;
end;

```

The first optimization is to have the function compiled into native code by inserting the keyword `native`.

Because this function performs a number of integer operations, typing the variables is also straightforward. Argument `a` is an array; the local variables `low`, `high`, and `mid` are integers. The function with the `native` keyword and the type declarations looks like this:

```

func native (array a, k)
  begin
    local int low := 0, high := Length(a)-1, mid;
    while high >= low do
      begin
        mid := (low + high) div 2;
        if a[mid] > k then high := mid - 1
          else if a[mid] < k then low := mid + 1
            else return mid;
          end;
      nil;
    end;
  end;

```

As it is now, the function can be used to search arrays of anything that can be compared with the `<` and `>` operators. If you know you're searching for an integer in an array of integers, you can also type the `k` argument.

## Tuning Performance

Finally, you can look for optimizations in the code itself. Note, for example, that this function accesses `a[mid]` twice when `a[mid]` is less than or equal to `k`. You can save a little bit of time by putting `a[mid]` in a local variable.

The function with these last two optimizations in place looks like this:

```
func native (array a, int k)
begin
  local int low := 0, high := Length(a)-1, mid;
  local int value;
  while high >= low do
    begin
      mid := (low + high) div 2;
      value := a[mid];
      if value > k then high := mid - 1
        else if value < k then low := mid + 1
        else return mid;
    end;
  nil;
end;
```

In timings of the stand-alone functions searching an array of the numbers from 0 to 999, with `k` set to 501, the optimized function ran in one one-thousandth the time of the original function. Functions that manipulate symbolic data—copying strings or frames, for example—are unlikely to realize improvements of this magnitude through the use of the native compiler.

## Profiling Native Functions

---

The tracking itself adds a disproportionate amount of time to the execution of native functions called from other native functions. The profiler therefore gives you the choice of compiling native functions for accurate execution time or compiling them for detailed profiling.

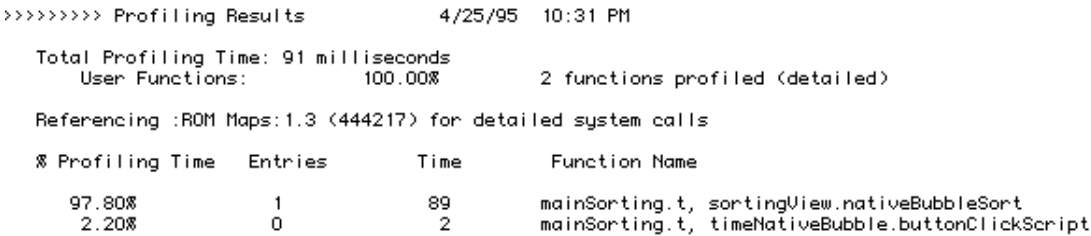
Tuning Performance

If you leave the Profile Native Functions option unchecked in Project Settings, the profile shows only the native functions that are called from interpreted functions—the time for each includes any time spent executing other native functions. If you enable that option, all native functions appear in the profile, but the times are distorted.

Consider, for example, the `bubbleSort` method used in “Marking Functions for Profiling” beginning on page 8-2. The `bubbleSort` method itself calls a function named `Swap`.

If the `Swap` method is also compiled native, and you have not checked the Profile Native Functions option, only the `bubbleSort` method appears in the profile, as illustrated in Figure 7-7.

**Figure 7-7** A profile of a native function calling another native function, without native-function profiling



If both methods are compiled native, and you have checked the Profile Native Functions option, both functions appear in the profile, as illustrated in Figure 7-8.

## Tuning Performance

**Figure 7-8** A profile of a native function calling another native function, with native-function profiling

```
>>>>>>> Profiling Results          4/25/95  10:33 PM

Total Profiling Time: 236 milliseconds
  User Functions:          100.00%      3 functions profiled (detailed)

Referencing :ROM Maps:1.3 (444217) for detailed system calls

% Profiling Time  Entries      Time      Function Name
49.58%           1          117      mainSorting.t, sortingView.nativeBubbleSort
49.58%          81          117      mainSorting.t, sortingView.nativeSwap
 0.85%           0           2       mainSorting.t, timeNativeBubble.buttonClickScript
```

The increased execution time reflects the use of the interpreter's function-call operation, which is necessary to make the call visible to the profiler. The impact is much like that of calling an interpreted function from a native function, as described in "Timing Interactions" beginning on page 8-16.

Tuning Performance



# NTK Commands

---

This chapter describes the commands available through the NTK menus.

## File Menu

---

You use the File menu to create, save, print, and otherwise manipulate files.

### New Layout (Ctrl-N)

---

Opens a new, untitled layout window.

You use this command to start a file to hold templates for your application's views. You name the file when you save it.

"Drawing, Resizing, and Moving Views" beginning on page 5-4 describes how you lay out views in a layout file. "Layout Files" beginning on page 4-3 describes how you use layout files in a project.

## NTK Commands

## New Proto Template (Ctrl-T)

---

Opens a new, untitled proto layout window.

You use this command to lay out your own protos. You name the file when you save it.

“Creating User Protos” on page 5-16 describes user protos.

## New Text File

---

Opens a new, untitled text file.

You use text files to hold an application’s install and remove scripts and any other NewtonScript code that’s outside the scope of the view templates.

“Text Files” beginning on page 4-31 describes how you use text files in a project.

## Open (Ctrl-O)

---

Opens a browser window on a saved layout file or a text-editing window on a saved text file. The Open command displays a dialog box through which you specify the file to be opened.

Shortcut: Double-click a file in the project window to open it for editing.

You open a project file the Project menu. You open layout windows and the Inspector window through the Windows menu.

## Link Layout

---

Brings an external layout file into the local hierarchy, by linking the external file to a linked subview template selected in a layout window. The Link Layout command displays the standard get-file dialog box for identifying the external layout file.

“Linking Multiple Layouts” beginning on page 5-14 describes linked layouts.

## NTK Commands

## Close (Ctrl-W)

---

Closes the active window and its associated file.

If you close a window whose file has been edited since it was last saved, NTK displays a dialog box that gives you a chance to save changes before closing the window.

## Save (Ctrl-S)

---

Saves the file associated with the active window. The saved file replaces the previously saved file of the same name. The file remains open.

The Save command affects only the one active window, that is,

- the layout file for the active layout window,
- the layout file for the active browser window,
- the project file associated with the project window
- the Inspector file, or
- the active text file.

## Save As

---

Saves a new copy of the file associated with the active window. The Save As command opens a file-save dialog box, through which you specify the new name and location. Save As changes the name of the active window and closes any open file with the window's previous name.

You use Save As to name or rename a layout file and to create a new file.

## Save All (Ctrl-M)

---

Saves all open NTK files, that is, any text or layout files, the project file, and the Inspector file. If you've not yet saved a file associated with one of the windows, NTK prompts you for the necessary filename.

## NTK Commands

## Revert

---

Restores the last saved version of the file associated with the active window. Any changes you've made since the last save are discarded.

## Print Setup

---

Displays the Print Setup dialog box, which lets you specify the paper size and page orientation for printing.

## Print One

---

Prints one copy of the file associated with the active window on the printer already selected through the Chooser. This command prints without displaying the Print dialog box for your verification.

## Print (Ctrl-P)

---

Displays the print dialog box, which identifies the selected printer and lets you verify or change the printing options. Clicking OK in the print dialog box triggers the printing of the current document according to the settings.

## Exit

---

Closes all open files and quits the Newton Toolkit. If you have made changes since saving any open files, NTK prompts you to save or discard the changes before closing the files.

## Recent File

---

Maintains a list of the four most recently opened files.

## Edit Menu

---

You use the Edit menu to manipulate the contents of a window—editing views in a layout window, for example, or text in a browser window.

### Undo (Ctrl-Z)

---

Cancels the last change made in the active window.

If NTK cannot undo the last operation, this command reads Can't Undo and is disabled. You cannot undo changes to a slot after you've applied them.

### Redo (Ctrl-A)

---

Reverts the change made by the previous Undo command. If NTK cannot redo the last operation, this command is disabled.

### Cut (Ctrl-X)

---

Deletes the current selection in the active window and places it on the Clipboard. You can then paste the material elsewhere. (Cut replaces anything previously copied or cut to the Clipboard.)

### Copy (Ctrl-C)

---

Places on the clipboard a copy of the current selection in the active window. You can then paste the material elsewhere. (Copy replaces anything previously copied or cut to the Clipboard.)

### Paste (Ctrl-V)

---

Pastes the contents of the Clipboard at the current insertion point.

## NTK Commands

If you're pasting into a layout window, Paste places the contents of the Clipboard inside the selected view.

### Clear (Delete)

---

Deletes the current selection without placing the material on the Clipboard.

### Duplicate (Ctrl-D)

---

Makes a copy of the currently selected view or views and places the copy in the same parent view.

### Shift Left

---

Shifts selected text or the line containing the insertion point one tab stop to the left.

### Shift Right

---

Shifts selected text or the line containing the insertion point one tab stop to the right.

### Select All (Ctrl-A)

---

Selects everything in the active window (that is, all views in a layout window, all text in an editor, or all files in a project window).

### Select Hierarchy

---

Selects all child views of the selected view or views in a layout window. Selection continues down the hierarchy to the last child view.

## NTK Commands

## Select in Layout

---

Selects the view in a layout window that corresponds to the currently selected view template in a browser. To use the Select in Layout command, first select a template in a browser template list.

Shortcut: Double-click on a template name in a browser window to select the corresponding view in an open layout window for the same file.

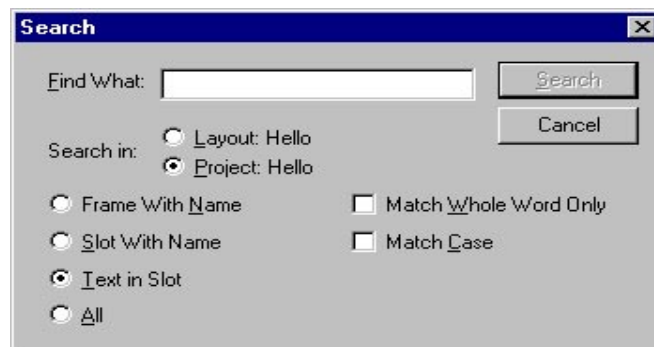
You can also use Select in Layout to select invisible or hidden views.

## Search (Ctrl-R)

---

Finds and lists instances of a string in the active layout file or in all layout files in a project. The Search command displays a dialog box, illustrated in Figure8-1, in which you specify the string you want to find and select search specifications.

**Figure8-1** The dialog for searching with Search



"Searching Template Files" beginning on page 5-25 describes the settings in the Search dialog box.

## NTK Commands

## Find (Ctrl-F)

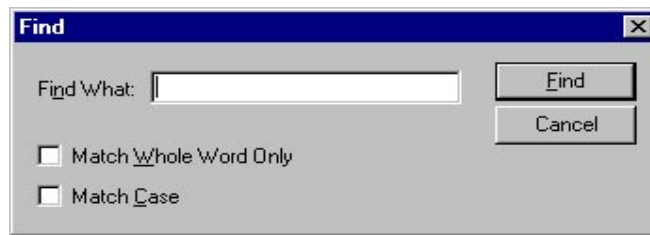
---

Finds a string in the active text window.

The Find command displays a dialog box, illustrated in Figure0-1, in which you specify the text you're interested in and a pair of search specifications.

---

**Figure0-1** The dialog for searching with Find



The Find command is available only when a text window is active or a text slot is open for editing.

“Searching the Active Window” beginning on page 5-26 describes the Find dialog box.

## Find Next (Ctrl-G)

---

Finds the next instance of the last string specified through the Find command, which is documented in “Searching the Active Window” beginning on page 5-26.

## Find Inherited

---

Finds the first occurrence of the currently selected slot in the parent view template hierarchy. The Find Inherited command is available only when a browser window is active.



## NTK Commands

The Find Inherited command looks first in the parent of the selected template. If NTK doesn't find the selected slot there, the search continues up the parent hierarchy to the file's layout view (that is, NTK does not search across linked layouts). When it finds a slot with the same name as the selected slot, NTK opens another browser window, with the slot and its template selected. If it doesn't find the slot in any template in the local hierarchy, NTK sounds the system beep.

## Newt Screen Shot

---

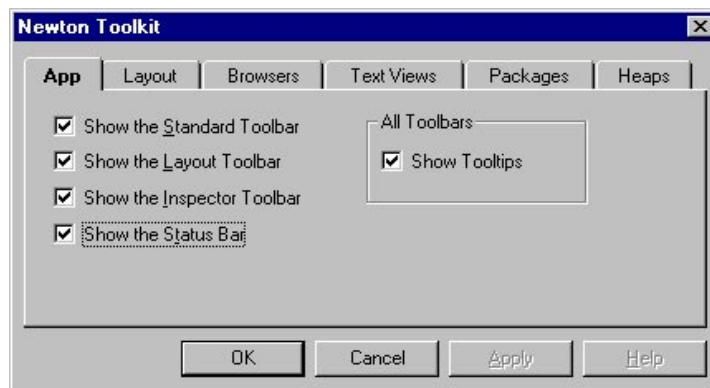
Places a bitmap of the Newton screen on the Clipboard. This item is available only when the Inspector is connected.

## Toolkit Preferences

---

Displays the Toolkit Preferences dialog box, illustrated in Figure8-2, through which you set various characteristics of the Toolkit.

**Figure8-2** The App preferences of the Toolkit Preferences dialog box

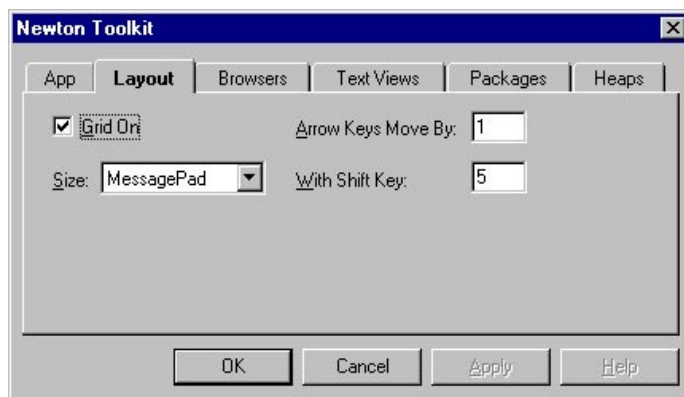


## NTK Commands

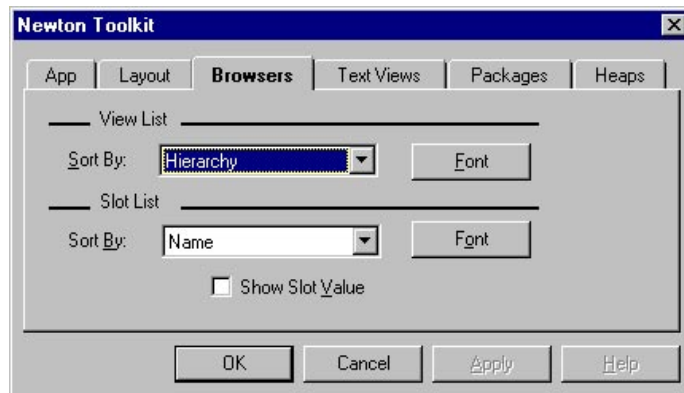
When you click OK, NTK stores your current settings in a file named Newton Toolkit Preferences in the system Preferences directory.

The fields in the Toolkit Preferences dialog box are described in “Toolkit Preferences” beginning on page 4-19.

**Figure8-3** The Layout preferences of the Toolkit Preferences dialog box

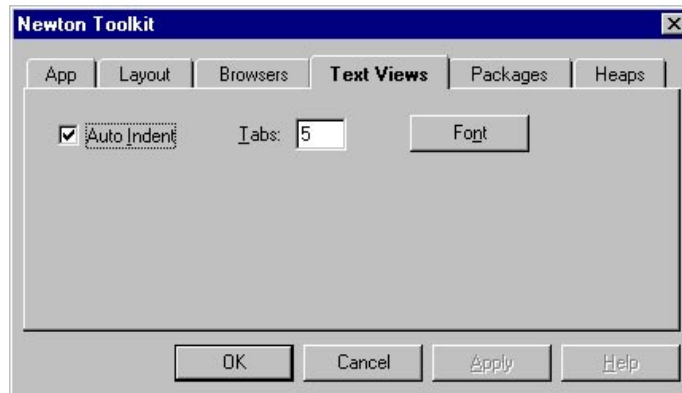


## NTK Commands

**Figure8-4** The Browsers preferences of the Toolkit Preferences dialog box

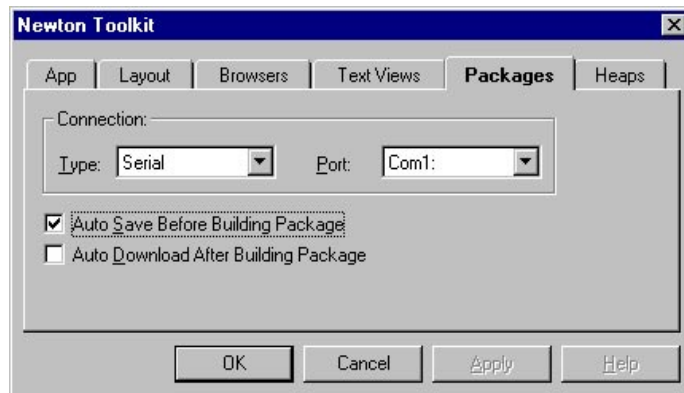
## NTK Commands

**Figure8-5** The Text Views preferences of the Toolkit Preferences dialog box



## NTK Commands

**Figure8-6** The Packages preferences of the Toolkit Preferences dialog box



## NTK Commands

**Figure8-7** The Heaps preferences of the Toolkit Preferences dialog box



## Project Menu

---

You use the project menu to manage your application project.

### New Project

---

Creates and opens a new project file and opens a project window.

The project file contains project specifications and the list of files that make up the project, that is, the files that NTK processes during the build.

“Project File” beginning on page 4-2 describes the project file.

### Open Project

---

Opens an existing project file. You can have only one project open at a time.

### Add Window

---

Adds to the project the file associated with the active window.

### Add File

---

Adds a file to the project from anywhere on the desktop. The Add File command displays a dialog box through which you specify the name and location of the file.

### Remove File

---

Removes the selected file or files from a project.

## NTK Commands

## Update Files

---

Verifies that all entries in the project file can be resolved to files that currently exist. When it can't resolve an entry, NTK displays a dialog box, through which you identify the correct target file.

You use Update Files to update the project file when you've moved or renamed a file since adding it to a project.

## Build Package (Ctrl-1)

---

Builds a project—usually a package—from the files and specifications in the open project file.

NTK places the package file in the same folder as the project file. The name of the package file is the name of the project with the suffix `.pkg`.

If the Output option in the Output Settings dialog box is set to Stream file, NTK builds an object stream file and places it in a file with the name of the project and the suffix `.stream`.

You can rename the output file.

The section “Building a Project” beginning on page 4-28 describes how NTK builds a project.

## Download Package (Ctrl-2)

---

Downloads the package file for the open project to a Newton device.

You must install the Toolkit application on the Newton, as described in Chapter 1, “Installation and Setup,” before you can download a package to it.

## Export Package to Text

---

Writes the contents of the project data file and all files in a project into a text file. The name of the file is the project name with the suffix `.txt`.

You can open this file in any application that recognizes text files.



## NTK Commands

## Install Toolkit App

---

Installs the Toolkit application on a Newton PDA connected to the development system.

The Toolkit application handles the downloading of packages and supports the Inspector and performance profiler.

Chapter 1, “Installation and Setup,” describes how to set up a connection between the development system and the PDA prior to downloading the Toolkit application.

## Mark as Main Layout

---

Designates the selected file in the project window as the main layout file—that is, the layout file whose base view is the application base view.

This item applies only to application projects, that is, projects configured to produce a new part of type `form`.

## Process Earlier (Ctrl-Up Arrow)

---

Moves the selected file in the project window one place closer to the beginning of the build list.

## Process Later (Ctrl-Down Arrow)

---

Moves the selected file in the project window one place closer to the end of the build list.

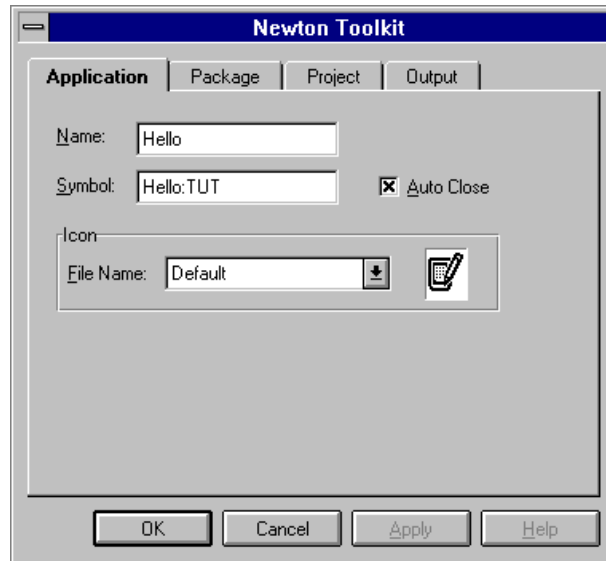
## Settings

---

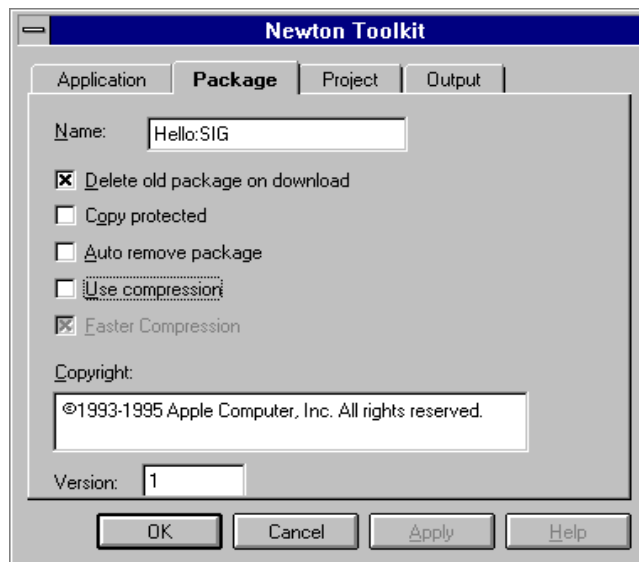
Displays the Settings dialog box, illustrated in Figure 8-8, through which you establish the application, package, project, and output specifications described in “Project Settings” beginning on page 4-12.

## NTK Commands

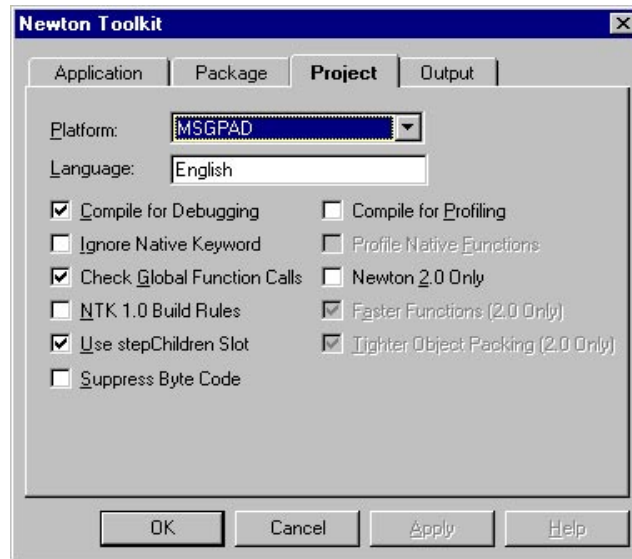
**Figure8-8** The Application Settings panel of the Settings dialog box

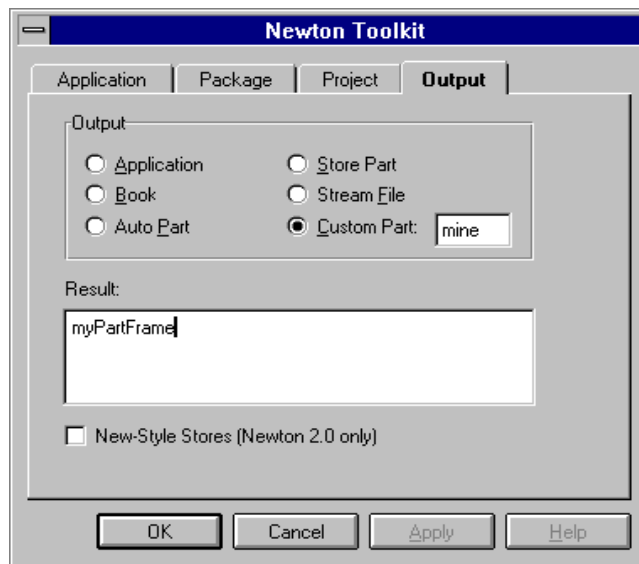


## NTK Commands

**Figure8-9** The Package Settings panel of the Settings dialog box

## NTK Commands

**Figure 8-10** The Project Settings panel of the Settings dialog box

**Figure8-11** The Output Settings panel of the Settings dialog box

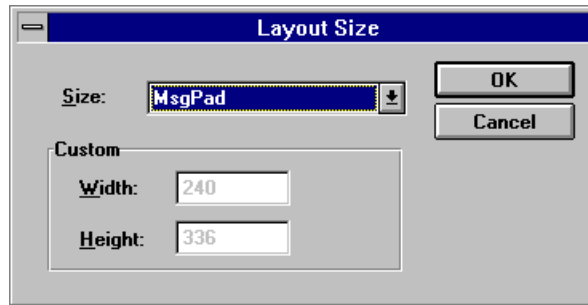
## Layout Menu

You use the layout menu to control the layout environment and manipulate views in the layout window.

## Layout Size

Displays a dialog box, illustrated in Figure8-12, that lets you set the size of the layout.

**Figure8-12** The Layout Size dialog box



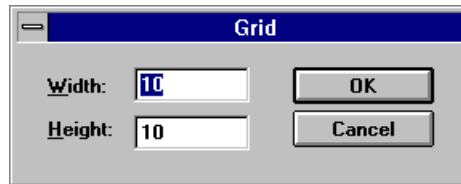
If you choose a platform from the Platform drop list, NTK fills in the width and height from the information in the platform file. If you choose Custom from the drop list, NTK lets you set your own values for the width and height of the layout in pixels.

## Autogrid On

Turns Autogrid on and off. Autogrid constrains the corners of views to nodes on the grid. The default grid resolution is 8 by 8 pixels. You can change the resolution through the Set Grid command.

## Set Grid

Opens the dialog box, illustrated in Figure8-13, that lets you change the grid size used with Autogrid. The units are pixels.

**Figure8-13** The Set Grid dialog box

---

## Move To Front

Moves the selected view in front of its siblings on the screen by placing it behind its siblings in the drawing list.

“Ordering Views” on page 5-10 describes how views are ordered.

---

## Move Forward (Ctrl-Down Arrow)

Moves the selected view one step later in the drawing list, so that it’s drawn after the view it previously preceded.

“Ordering Views” on page 5-10 describes how views are ordered.

---

## Move To Back

Moves the selected view behind its siblings by placing it ahead of its siblings in the drawing list.

“Ordering Views” on page 5-10 describes how views are ordered.

---

## Move Backward (Ctrl-Up Arrow)

Moves the selected view one step earlier in the drawing list, so that it’s drawn before the view it previously followed.

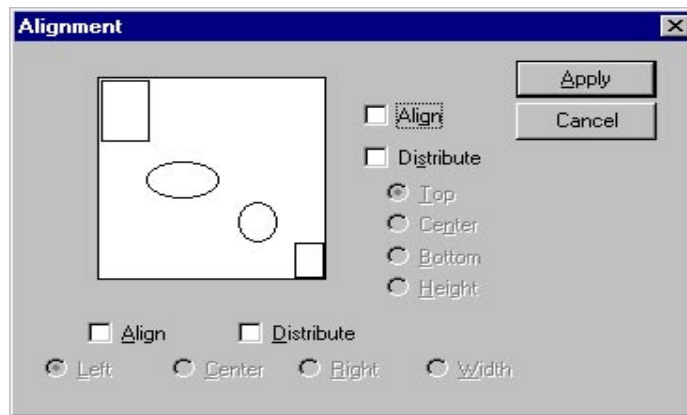
“Ordering Views” on page 5-10 describes how views are ordered.

## NTK Commands

## Alignment

Opens the dialog box, illustrated in Figure8-14, through which you establish a view alignment scheme that's applied to the selected views when you click Apply or subsequently choose Align.

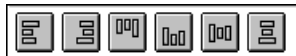
**Figure8-14** The Alignment dialog box



“Aligning Views” beginning on page 5-8 describes the Alignment dialog box.

A subset of the alignment options, illustrated in Figure8-15, appears on the drawing palette.

**Figure8-15** The alignment buttons on the palette





## NTK Commands

## Align

---

Aligns the selected views using the alignment scheme specified through Alignment.

## Preview (Ctrl-Y)

---

Toggles the layout screen between layout mode and preview mode. In Layout mode, NTK shows the rectangular extents of each view on the screen. In preview mode, NTK displays the views approximately as they would appear on the Newton screen.

“Previewing” beginning on page 5-11 describes preview mode.

## Browser Menu

---

You use the Browser menu to manipulate slots and to control how the browser displays templates and slots.

## Template Info (Ctrl-I)

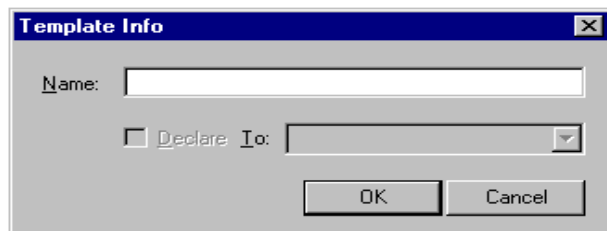
---

Opens a dialog box in which you can name and declare a selected view. A view name operates as a symbol in NewtonScript. Declaring a view allows you to access it symbolically from the view in which it's declared and from descendants of that view.

Figure8-16 illustrates the Template Info dialog box.

“Naming and Declaring Views” beginning on page 5-13 describes the Template Info dialog box.

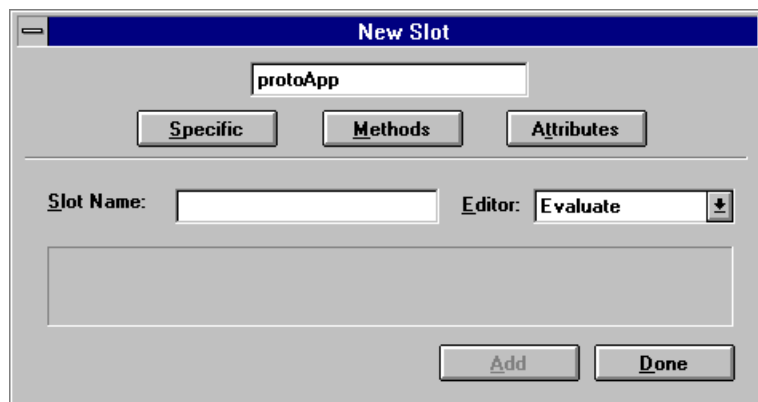
---

**Figure8-16** The Template Info dialog box, for naming and declaring views

## New Slot

Opens a dialog box, illustrated in Figure8-17, for adding new slots to the selected template.

---

**Figure8-17** The New Slot dialog box

“Adding Slots” beginning on page 5-18 describes the New Slot dialog box.

## NTK Commands

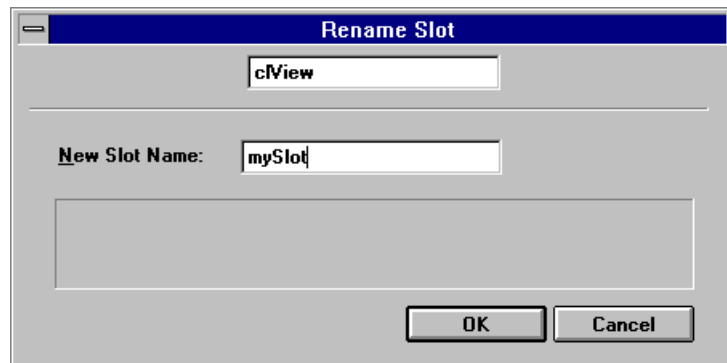
## Rename Slot

---

Opens a dialog box, illustrated in Figure8-18, that lets you rename the selected slot.

---

**Figure8-18** The Rename Slot dialog box



If the slot is open for editing, Rename Slot applies outstanding changes and closes the slot editor before changing the name.

Note that changing the name of the slot does not change existing occurrences of the name in scripts. Changing the case of a name through Rename Slot changes the name.

## Templates By Type

---

Lists templates alphabetically by type.

## Templates By Hierarchy

---

Lists templates by hierarchy, with sibling views listed in the order they're created.

NTK Commands

## Slots By Name

---

Orders slots alphabetically in the browser slot list.

## Slots By Type

---

Orders slots by type in the browser slot list.

## Show Slot Values

---

Displays the value of each slot next to the slot name in the browser slot list.

## Apply (Ctrl-E)

---

Checks syntax and inserts into the slot changes made in a slot editor.

## Revert

---

Discards any changes made since the last Apply or Save.

## Window Menu

---

You use the Window menu to open browser windows and to open the Inspector window or connect the Inspector.

### Open Inspector

---

Opens the Inspector without making a connection to a Newton device. You can have only one Inspector window open at a time.

### Connect Inspector (Ctrl-K)

---

Connects the Inspector to a Newton PDA with the Toolkit application installed and a connection to the development system. If the Inspector window is not open, Connect Inspector opens it.

Chapter 1, “Installation and Setup,” describes how to set up a connection between the development system and the Newton and how to install the Toolkit application. Chapter 6, “Debugging” describes the commands available through the Inspector.

### New Browser (Ctrl-B)

---

Opens a new template browser on a layout file selected in a project window or at the level of the selected view in a layout window.

You use the browser to edit templates in a layout file. “Browsing and Editing Templates” beginning on page 5-16 describes the browser.

### Open Layout (Ctrl-L)

---

Opens a layout window for the layout file selected in the project window.

NTK Commands

## Cascade

---

Arranges windows in cascade fashion.

## Tile

---

Arranges windows in tile fashion.

## Arrange Icons

---

Arranges the icons within the selected window.

## Set Default Window Position

---

Sets the default window size for projects, layouts, browsers and text files.

# Help Menu

---

You use the help menu for assistance with specific NTK Index topics.

## Index

---

Opens Help window which displays NTK Index of available Help items.

## Command Reference

---

Opens Help window which displays the commands available through the NTK Index.

## Using Help

---

Opens Help window which displays Using Help items.

NTK Commands

## About Newton Toolkit

---

Displays information on current version of NTK and Product Team.

NTK Commands



# Keyboard Text-Editing Commands

---

This appendix lists the keyboard commands for navigating and manipulating text in NTK slots, the project data file, and the Inspector window.

You can use the keyboard to

- specify an insertion point
- select text
- manipulate selected text
- delete text
- change the effect of the next keystroke

The keyboard text-editing commands operate relative to the current insertion point or the selected text. “Editing Text” beginning on page 5-23 describes the basic NTK text editor.

You can reverse the last keyboard command by pressing Ctrl-Z (Undo).

## Setting the Insertion Point

---

You can set the insertion point with the commands in Table A-1. If text is selected when you set the insertion point, it is simply deselected.

## Keyboard Text-Editing Commands

**Table A-1** Moving the insertion point

<b>Motion</b>	<b>Keystrokes</b>
Back one character	Left arrow
Forward one character	Right arrow
Down one line	Down arrow
Up one line	Up arrow
To beginning of word, or back one word	Ctrl-Left arrow
To end of word, or to end of next word	Ctrl-Right arrow
To beginning of line	Home
To end of line	End
To next page	Page down
To previous page	Page up
To beginning of text	Ctrl-Home
To end of text	Ctrl-End

## Keyboard Text-Editing Commands

## Selecting Text

---

You can select text with the commands listed in Table A-2. If text is already selected when you issue one of the text-selection commands, the selection is extended.

**Table A-2**      Selecting text with keyboard commands

---

<b>Selection</b>	<b>Keystrokes</b>
One character back	Shift-Left arrow
One character forward	Shift-Right arrow
One word back	Ctrl-Shift-Left arrow
One word forward	Ctrl-Shift-Right arrow
Previous line	Shift-Up arrow
Next line	Shift-Down arrow

Indent selection right

Keyboard Text-Editing Commands

# Manipulating Selected Text

---

You can use the commands listed in Table A-3 to manipulate selected text. As the table shows, NTK supports the customary commands for cutting (Ctrl-X), copying (Ctrl-C), and pasting (Ctrl-V).

**Table A-3**      Manipulating selected text

---

Manipulation	Keystrokes
Cut selection to Clipboard	Ctrl-X
Copy selection to Clipboard	Ctrl-C
Paste contents of Clipboard over selection	Ctrl-V
Indent selection left	Ctrl-[
Indent selection right	Ctrl-]

# Deleting Text

---

You can delete text with the keyboard commands listed in Table A-4. The NTK keyboard editing commands do not place deleted text onto the Clipboard; to delete the selection and place it on the Clipboard, use Command-X.

**Table A-4**      Deleting text with keyboard commands

---

Scope of deletion	Keystrokes
Selection only	Delete Backspace
Selection or one character back	Backspace Shift-Backspace
Selection or one character forward	Delete forward

## Keyboard Text-Editing Commands

# Keyboard Shortcuts

---

This appendix lists keystroke combinations that invoke NTK menu items or and that affect views in a layout or Browser window.

**Table B-1** Keyboard equivalents to menu items

---

Key combination	Effect
Ctrl-N	New Layout
Ctrl-T	New Proto Template
Ctrl-O	Open
Ctrl-S	Save
Ctrl-M	Save All
Ctrl-P	Print
Ctrl-Z	Undo
Ctrl-X	Cut
Ctrl-C	Copy
Ctrl-V	Paste
Ctrl-D	Duplicate
Ctrl-A	Select All
Ctrl-R	Search
Ctrl-F	Find
Ctrl-G	Find Next
Ctrl-1	Build Package
Ctrl-2	Download Package

## Keyboard Shortcuts

**Table B-1** Keyboard equivalents to menu items (continued)

Key combination	Effect
Ctrl-Y	Preview
Ctrl-I	Template Info
Ctrl-E	Apply
Ctrl-Down arrow	Move Forward Process Later
Ctrl-Up arrow	Move Backward Process Earlier
Ctrl-K	Connect Inspector
Ctrl-B	New Browser
Ctrl-L	Open Layout

The keyboard commands listed in Table B-1 move templates within the view hierarchy. You can issue these commands with a template selected in a browser window. When a view is selected in a layout window, the Ctrl key in combination with the arrow keys resizes the view, as described in “Resizing a View” beginning on page 5-7.

**Table B-2** Keyboard commands that affect the hierarchy

Key combination	Effect
Ctrl-Left arrow	Move a template one layer up in the hierarchy
Ctrl-Right arrow	Move a template one layer down in the hierarchy
Ctrl-Up arrow	Move a template one place up in the drawing list (that is, move view backward).
Ctrl-Down arrow	Move a template one place down in the drawing list (that is, move view forward)



## A P P E N D I X B

### Keyboard Shortcuts

## A P P E N D I X B

### Keyboard Shortcuts

# Custom Bitmaps and Sounds

---

A **bitmap**, “BMP”, or **sound**, “WAV”, is a piece of data stored on the development system and incorporated into a Newton application during the project build. You can use bitmaps, sound, and data from other sources in your application. NTK explicitly supports 'BMP' files and 'WAV' files, and it includes functions for converting other files into resources your application can handle.

This appendix describes how to embed 'BMP' and 'WAV' files in Newton applications.

## Adding Bitmap and Sound Files to a Project

---

You add a bitmap or sound file to a project through the Add File item in the File menu.

## Using Bitmap and Sound Files

---

You can include 'BMP' files and 'WAV' files in an application. For example, to include WAV files, you must

1. extract sound data from files and convert to sound format
2. embed sound data in to application

## Opening and Closing Resource Files

---

NTK automatically opens and closes the resource files containing 'BMP' files that you access through the picture slot editor or the Settings dialog box. You need to open and close resource files only if you're manipulating the resources directly—when you're using sound files, for example, or when you're using data specific to your application.

## Using the Resource-Handling Functions

---

Current versions of NTK and the Newton object system support some kinds of resources more fully than others.

The use of 'BMP' resources is well supported in the Newton object system and NTK. For many of the system-supplied view prototypes, NTK locates the appropriate resource automatically when an external resource file is included in the application's NTK project file.

On the other hand, the object system does not currently supply any prototypes that use external sound resource files; thus, you need to do a little more work to incorporate them in your application.

This section describes how NTK handles `bitmap` and sound files and describes data extraction functions that are specialized for these resource types.

### Using Bitmaps

---

You can draw your pictures in any graphics program, and then paste them as 'BMP' files. You add the bitmap file to an NTK project through the Add File item in the Project menu.

NTK lets you add named 'BMP' files to picture slots in your templates through the standard picture slot editor, illustrated in Figure C-1

## Custom Bitmaps and Sounds

**Figure C-1** Adding a named 'BMP' file to a picture slot

The File drop list contains all bitmap files that have been added to the project file. The Picture list shows all named 'BMP' files. For more information on the picture slot editor, see “Editing Slots” beginning on page 5-20.

## Making a Bitmap From a 'BMP' File

NTK also supplies the `GetBMPAsBits` function for extracting bitmaps from 'BMP' files. NTK itself uses this function when manipulating the files you access through the picture slot editor.

The `GetBMPAsBits` function retrieves a 'BMP' file by name, converts the 'BMP' to a bitmap, and returns a frame containing a bitmap object. It accepts as its arguments the name of the 'BMP' file to be retrieved and a Boolean value specifying whether to retrieve the mask for the bitmap from the file. This function is described completely in “GetBMPAsBits” beginning on page C-9, in the reference section of this appendix.

The following code example retrieves the 'BMP' file named `Daphne` and stores it in the compile-time variable `gDaphBitd`.

```
gDaphBits := GetBMPAsBits(HOME & "Daphne.BMP", nil);
```

You can make the bitmap data available at run time by storing it in an evaluate slot.

## Custom Bitmaps and Sounds

The Drawing and Graphics chapter in *Newton Programmer's Guide: System Software* illustrates how you can use bitmap data when drawing.

## Using External Sound Files

---

You can use any program that saves 'WAV' files to create sounds for your application. NTK supplies the function, `GetWAVAsSamples` for extracting sound data from these resources.

The `GetWAVAsSamples` function reads a sound sampled at 22kHz and returns a Newton sound frame. The function expects as its sole argument a string specifying the name of the sound file.

The following code example retrieves by name the 'WAV' file `chickadee` and stores it in the compile-time variable `gChickadee`. To make sound data available to the application, you create a compile-time global `gChickadee` to initialize an evaluate slot in the application's base view.

```
gChickadee := GetWAVAsSamples("chickadee");
```

You can make the sound available at run time by storing it in an evaluate slot.

See the Sound chapter in *Newton Programmer's Guide: System Software* for more complete information on using sound in Newton applications.

## Custom Functions

---

This section describes the functions used to make frame objects from data files. These functions are available only during compile time—they are not available at run time.

### Retrieving Resources

---

This section documents the functions to use to retrieve resources from an open resource file. You can retrieve resources by type and either name or resource ID.

## Custom Bitmaps and Sounds

**GetBMPAsBits**

---

```
GetBMPAsBits(nameString, maskToo)
```

Retrieves the specified 'BMP' resource by name from an open resource file, converts the 'BMP' to a bitmap, and returns a frame containing a bitmap object and an optional mask.

<i>nameString</i>	A string specifying the name of the resource to be retrieved.
<i>maskToo</i>	<p>A Boolean value indicating whether to include a mask in the returned frame. A mask is a companion bitmap used for highlighting a screen element. If <i>maskToo</i> is non-nil, a mask is obtained by one of two means:</p> <p>First, <code>GetBMPAsBits</code> looks in the resource file for a resource with the same name as the specified 'BMP' resource but with an exclamation point appended. If the resource is found, it is returned in the mask slot.</p> <p>Second, if no mask resource is found, a resource is automatically constructed and returned in the mask slot of the bitmap object.</p> <p>If the <i>maskToo</i> parameter is nil, no mask is found or constructed for the bitmap.</p>

The bitmap object returned by this function is a frame with the following slots:

<i>bits</i>	A reference to a binary object containing the bitmap data
<i>bounds</i>	<p>A bounds frame specifying the dimensions of the bitmap; for example,</p> <pre>{left: 0,   top: 0,</pre>

## Custom Bitmaps and Sounds

```
right: bitmapWidth,
bottom: bitmapHeight}
```

*mask* A reference to a binary object containing the mask bitmap. This slot is included only if the *maskToo* argument was not `nil`.

**Note**

Picture objects are stored much more compactly as binary 'BMP' objects than as bitmap objects (obtained with `GetBMPAsBits`). Drawing from a bitmap, however, may be significantly faster. The Drawing and Graphics chapter of *Newton Programmer's Guide: System Software* contains more discussion of picture objects. ♦

**GetWAVAsSamples**

---

```
GetWAVAsSamples(filename)
```

Retrieves the specified 22KHz sound resource from the currently open resource file and returns the data in Newton sound format.

*nameString* A string specifying the name of the sound file to be retrieved.

**LoadDataFile**

---

```
LoadDataFile(filename, class)
```

Reads arbitrary data stored in the file named and returns the result as a binary object with the specified class.



## Summary of Custom Functions

---

This section categorizes the resource-manipulation functions by task.

### Getting Custom Data

---

`GetBMPAsBits(filename, maskToo)`

`GetWAVAsSamples(filename)`

`LoadDataFile(filename, class)`

## A P P E N D I X C

### Custom Bitmaps and Sounds

# Specialized Slot Editors

---

This appendix describes the specialized slot editors you use for editing the system-defined slots. The description of the `viewBounds` slot, which is a simple rectangle slot, articulates the meanings of the four integers under different justification settings.

## Script Slots

---

You edit the slots containing system-defined messages with the basic NTK text editor described in “Editing Text” beginning on page 5-23.

The system messages appear in the Specific and Methods pop-up menus in the browser and New Slot dialog boxes. When you add one of these slots, NTK places the skeletal structure of the method in the slot. If you add a `viewStrokeScript` slot, for example, NTK defines the initial slot contents as

```
func(unit)
begin
end
```

If a method takes no parameters or requires no special return value, NTK sets the initial contents to the simple function statement

```
func( )
begin
end
```

The system-defined messages are described in in the *Newton Programmer's Guide*.

## View Attributes

---

The view attributes slots contain various specifications that the Newton uses to create, display, and manipulate views. Some of the slots contain a single value or string. The `viewOriginX` and `viewOriginY` slots, for example, each contain a number, which you edit through the number editor. The `viewFont` slot contains a single statement that specifies a font name. You edit it and the other attribute slots containing text with the standard NTK text editor, described in “Editing Text” beginning on page 5-23.

This section illustrates the specialized editors you use to use to edit the more complex view attributes slots. For detailed descriptions of the fields, see the “Views” chapter in *Newton Programmer’s Guide: System Software*.

### viewBounds

---

Left: <input type="text" value="0"/>	Right: <input type="text" value="0"/>	Width: 0
Top: <input type="text" value="0"/>	Bottom: <input type="text" value="0"/>	Height: 0

The `viewBounds` slot defines the bounds of a view. NTK automatically fills in the `viewBounds` values when you lay out a view in the graphical editor. The values in the four slots are relative to the parent or sibling view, and the exact meaning varies with different justification strategies, as defined in the `viewJustify` slot. Table D-1 summarizes the meanings of the Left and Right fields with different horizontal view justification settings. Positive numbers are offset to the right, negative to the left. Table D-2 summarizes the meanings of the Top and Bottom fields with different vertical view justification settings. Positive numbers are offset down, negative up.

## Specialized Slot Editors

**Table D-1**      Meaning of viewBounds fields for horizontal justification

<b>Justification</b>	<b>Meaning of Left</b>	<b>Meaning of Right</b>
Left Relative	The offset from the parent's or sibling's left edge to the view's left edge.	The offset from the parent's or sibling's left edge to the view's right edge.
Right Relative	The offset from the parent's or sibling's right edge to the view's left edge.	The offset from the parent's or sibling's right edge to the view's right edge.
Center Relative	The left offset of the view's center from the parent's or sibling's center.	The total width of the view.
Full Relative	The offset of the view's left edge from the parent's or sibling's left edge.	The offset of the view's right edge from the parent's or sibling's right edge.

**Table D-2**      Meaning of viewBounds fields for vertical justification

<b>Justification</b>	<b>Meaning of Top</b>	<b>Meaning of Bottom</b>
Top Relative	The offset from the parent's or sibling's top edge to the view's top edge.	The offset from the parent's or sibling's top edge to the view's bottom edge.
Bottom Relative	The offset from the parent's or sibling's bottom edge to the view's top edge	The offset from the parent's or sibling's bottom edge to the view's bottom edge.
Center Relative	The vertical offset of the view's vertical center from the parent's or sibling's center.	The total height of the view
Full Relative	The offset of the view's top edge from the parent's or sibling's top edge.	The offset of the view's bottom edge from the parent's or sibling's bottom edge.

Specialized Slot Editors

viewFlags

☒ vVisible

☐ vApplication

☐ vCalculateBounds

☐ vReadOnly

☐ vClipping

☐ vFloating

☐ vWhiteSpaceProtected

☐ vNoScripts

Entry Flags

Field Type: 

None

☐ vSingleUnit

☒ vClickable

☐ vStrokesAllowed

☐ vGesturesAllowed

☐ vPunctuationAllowed

☐ vAnythingAllowed

☐ vCharsAllowed

☐ vLettersAllowed

☐ vMathAllowed

☐ vNumbersAllowed

☐ vShapesAllowed

☐ vCustomDictionaries

☐ vCapsRequired

viewFormat

Pen:

Roundness:

Inset:

Shadow:

Frame: 

None

Fill: 

None

Lines: 

None

viewJustify

View Position

Horizontal: Parent: 

Left Relative

Sibling: 

None

Vertical: Parent: 

Top Relative

Sibling: 

None

Text/Graphics

Horizontal: 

Left

Vertical: 

Top

Text Limits: 

No limit

Printing

☐ Reflow

☐ Lasso Children

D-4

View Attributes

Draft. Preliminary, Confidential. ©1996 Apple Computer, Inc.

## Specialized Slot Editors

viewEffect

---

Effect:  ▾

Steps:  Time:

Columns:  Rows:

☐ Alt. Horz. Columns ☐ Alt. Horz. Rows  
☐ Alt. Vert. Columns ☐ Alt. Vert. Rows

Horz. Dir.:  ▾ Vert. Dir.:  ▾

☐ Reveal Line ☐ Wipe ☐ From Edge

viewTransferMode

---

Transfer Mode:  ▾

## Specific Slots

---

The slots in the Specific pop-up menu represent the slots that are specific to the selected proto. These slots hold methods or simple values that you edit with one of the standard slot editors.

## A P P E N D I X D

### Specialized Slot Editors



# Newton Debugging Applications

---

This appendix describes a handful of Newton applications that help you test and examine your software.

NTK is shipped with a number of small debugging tools:

- HeapShow, which displays heap statistics while they're happening and lets you force low-memory conditions
- Snarf, which adds a simulated transport for testing communication software
- Exception Printer, which adds more information to exception reports on the Newton
- vFlags, which lets you manipulate the recognition flags for a clEdit view and test the effect on input recognition

The vFlags application is shipped with its source code, so you can modify the application for your own purposes.

The Newton Debugging Tools folder also contains a project named NSDShortCuts, which lets you manipulate your own debugging environment.

The bulk of this appendix is the HeapShow documentation.

## Installing the Debugging Packages

---

The Newton debugging applications are shipped as package files, which you can install on the Newton using the Newton Package Installer.

## Newton Debugging Applications

To remove an application from a Newton 2.0 unit, scrub its icon in the Extras drawer. To remove an application from a Newton MessagePad, use the Remove Software option in the Prefs application.

## HeapShow

---

This section describes the HeapShow application, which allows you to examine heap use on the Newton.

### About HeapShow

---

HeapShow is a Newton application that displays statistics about the Newton heaps—that is, the portions of Newton memory allocated for storing pointers, handles, and frames—in a floating view on the Newton screen.

While HeapShow is running, you can start up and use other applications and then watch the impact on the heaps.

### About Newton Memory Management

---

Memory is allocated in the Newton system in a number of ways, but most memory allocations are for either heaps or stacks. HeapShow lets you monitor

- the two biggest heaps: the pointers heap and the handles heap
- the frames “heap,” which is actually a large pointer allocation within the pointers heap
- the amount of unallocated memory

Heaps grow and shrink only as needed. If a memory allocation cannot be accommodated by the free space available in the target heap, then the heap grows in 1 KB increments until there’s enough contiguous space available. You can watch the changes in HeapShow: If the target heap is the pointers heap, for example, then the size of the pointers heap grows, and the amount

## Newton Debugging Applications

of free system memory shrinks. The amount of free space within the pointers heap might change.

When there's no more space to grow the size of the heap, the stack manager asks the various tasks to reduce the size of their heaps to free up memory—heaps and stacks free space only when asked, which is why a memory allocation in one heap can reduce the size of the pointers heap.

C-code often creates heaps specific to its tasks. Communication tools, for example, typically allocate a separate heap. Moreover, C-code needs memory to hold its stacks. HeapShow cannot display statistics about special heap or stack allocations.

If a piece of C-code allocates a pointer without specifying a heap, the memory comes out of the pointers heap. If a piece of C-code allocates a handle without specifying a heap, the memory comes out of the handles heap.

Allocations in NewtonScript are always made in the frames “heap,” which is actually a pointer allocation within the pointers heap. The frames heap has its own heap manager and does not grow and shrink like the other heaps.

The frames heap manager deallocates memory (that is, garbage collects) only when there's not enough space for a frame allocation. HeapShow lets you force the frames heap manager to deallocate memory every time it check the system, so you can see the minimum space needed.

## Using HeapShow

---

You start up HeapShow by tapping its icon, illustrated in Figure D-1.

---

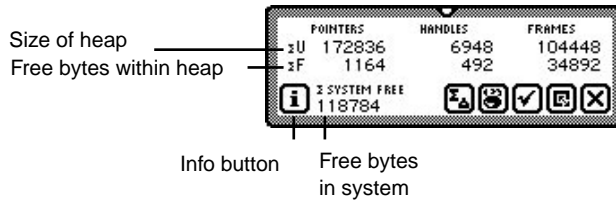
**Figure D-1**      The HeapShow icon



## Newton Debugging Applications

The HeapShow application displays the sizes and number of free bytes in the pointer, handles, and frames heaps, as illustrated in Figure D-2.

**Figure D-2** The default HeapShow display



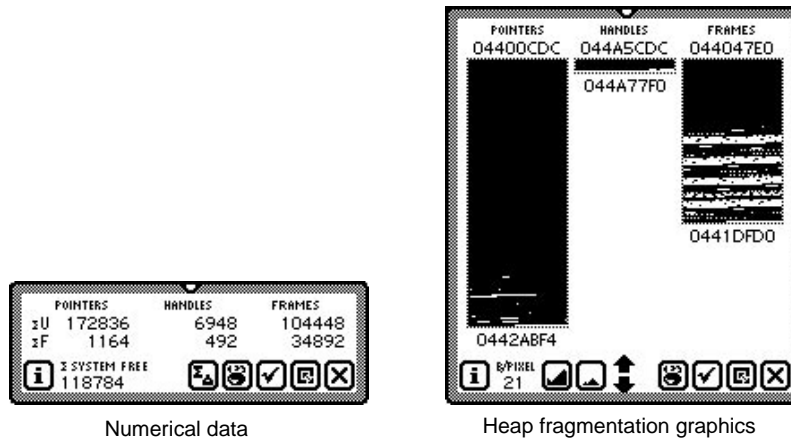
The Info button in the lower-left corner lets you set the preferences, which are described in “Preferences” beginning on page E-5.

The buttons along the lower-right edge, which are described in “HeapShow Controls” beginning on page E-8, let you

- control what information is displayed and how it’s presented
- force memory or statistics updates

## Statistics Display

HeapShow lets you examine the pointer, handles, and frames heaps on the Newton. You can adjust the display to show either numerical data or a graphical representation of the heap, as illustrated in Figure D-3.

**Figure D-3** Numerical data versus fragmentation graphics

You can also change the numerical display to show either

- the total sizes and number of bytes free in the three heaps or
- the differences in each since the display was last changed.

Note that the frames heap is of fixed size; only the number of free bytes changes.

“HeapShow Controls” beginning on page E-8 describes how to change the HeapShow display.

## Preferences

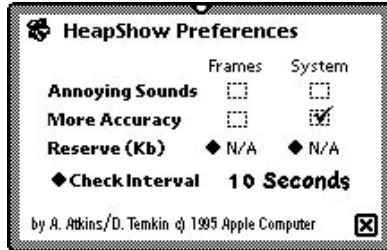
You can adjust the HeapShow Preferences to

- turn sound cues on and off
- balance the amount of data collected against the time spent collecting it
- set the interval at which HeapShow checks the status of the heaps.

To reach the Preferences settings, tap the Info button in the statistics display. Figure D-4 illustrates the HeapShow Preferences view.

# Newton Debugging Applications

**Figure D-4** HeapShow Preferences



You can adjust sound effects and accuracy independently for the heaps and for system memory.

**Annoying Sounds** Controls the HeapShow sound effects.

When Frames is checked, different sounds play if the amount of space used in either the pointer or handle heap grows or shrinks.

When System is checked, a sound plays if the availability of system memory changes.

**More Accuracy** Adjusts how thoroughly HeapShow researches the state of the heaps

When Frames is checked, HeapShow performs a garbage collection in the frames heap before reporting the statistics.

When System is checked, HeapShow includes the memory that stacks are willing to give back to the stack manager when calculating the system-wide free memory figure.

**Reserve (kB)** Allocates memory out of the frames heap, the pointer heap, or a newly created heap. You can use this option

## Newton Debugging Applications

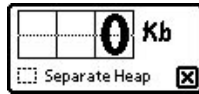
to create out-of-memory situations when testing your application.

When you tap one of the Reserve entries, HeapShow displays a view that lets you set the amount of memory to reserve.

If you set this number for Frames, HeapShow allocates the specified number of kilobytes in the frames heap. If you set this number for System, the memory comes out of the pointers heap. You can create a new heap and reserve it by activating the Separate Heap option, which appears in the Reserve System pop-up view, illustrated in Figure D-5.

Memory set aside by HeapShow is released when HeapShow exits.

**Figure D-5** Sizing the reserve pointers heap or a newly created heap

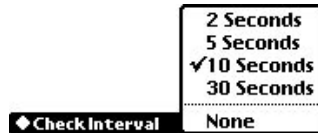


### Check Interval

Determines the interval at which HeapShow automatically checks memory statistics and updates the display. Tap the time field to access the list illustrated in Figure D-6.

## Newton Debugging Applications

**Figure D-6** Check Interval options

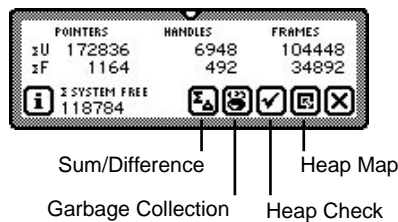


If you choose None, HeapShow updates the statistics only when you tap the Heap Check button, described on page E-9.

## HeapShow Controls

You tap the buttons on the lower-right edge of the HeapShow view to change the display and to force a garbage collection or a heap check. Figure D-7 illustrates the HeapShow controls.

**Figure D-7** The HeapShow controls



Tap the Sum/Difference button to toggle the display between

- the total sizes of all heaps and
- the size differences since the display was changed to show differences.

Tap the Garbage Collection button to force the Newton to reclaim unused memory.



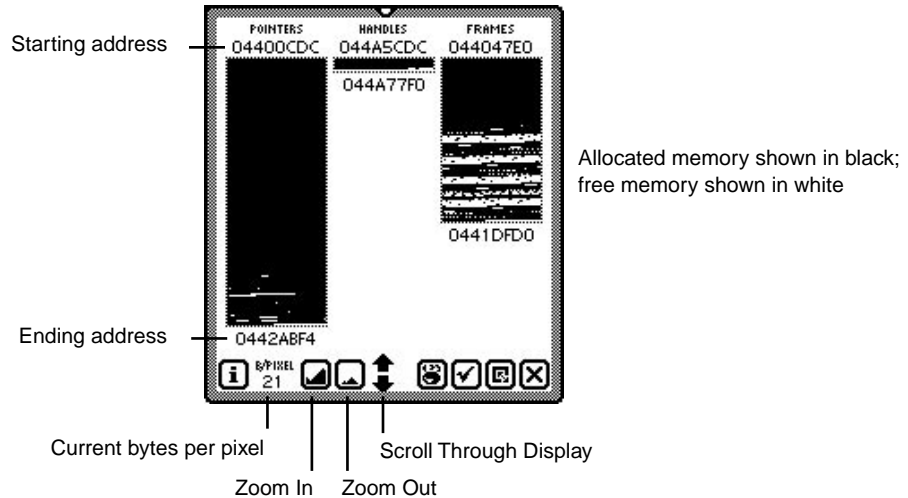
## Newton Debugging Applications

Tap the Heap Check button to force an immediate update of the stack statistics.

Tap the Heap Map button to toggle the display between

- a numerical presentation of the data and
- a display of heap maps that illustrate heap fragmentation

**Figure D-8** Heap fragmentation graphics



You can use the controls illustrated in Figure D-8 to zoom in or out and to scroll through the map.

## A P P E N D I X E

### Newton Debugging Applications

# Glossary

---

application base view

The topmost parent view in an application. The application base view encloses all other views that make up the application.

array

A sequence of numerically indexed slots (also known as the array elements) that contain objects. The first element is indexed by zero. Like other non-immediate objects, an array can have a user-specified class, and can have its length changed dynamically.

binary object

A sequence of bytes that can represent any kind of data, can be adjusted in size dynamically, and can have a user-specified class. Examples of binary objects include strings, real numbers, sounds, and bitmaps.

Boolean

A special kind of immediate value. In NewtonScript, there is only one Boolean, and it is called `true`. Functions and control structures use `nil` to represent false. When testing for a true/false value, `nil` represents false, and any other value is equivalent to `true`.

break loop	A state of the Newton processor in which program execution is suspended and the Newton accepts input only from an Inspector connection.
byte code	The hardware-independent instructions that are interpreted when a NewtonScript function executes.
child	A frame that references another frame (its parent) from a <code>_parent</code> slot. With regard to views, a child view is enclosed by its parent view.
class	A symbol that describes the data referenced by an object. Arrays, frames, and binary objects can have user-defined classes.
constant	A value that does not change. In NewtonScript the value of the constant is substituted wherever the constant is used in code.
declaring a template	Registering a template in another view (usually its parent) so that the template's view is pre-allocated when the other view is opened. This allows access to methods and slots in the declared view.
evaluate slot	A slot that's evaluated when NTK compiles the application.
flag	A value that is set either on or off to enable a feature. Typically flag values are single bits, though they can be groups of bits or a whole byte.
font spec	A structure used to store information about a font, including the font family, the font style, and the point size.
frame	An unordered collection of slots, each of which consists of a name and value pair. The value of a slot can be any type of object, and slots can be added or removed from frames dynamically. A frame can have a user-specified class. Frames can be used like records in Pascal and structs in C, but can also be used as objects which respond to messages.
function-call stack	A virtual stack that contains an activation record for each active function. See <b>stack activation record</b> .

function object	<p>An executable object in NewtonScript. Function objects are created by the NTK compiler from the function constructor:</p> <pre>func (args) funcBody .</pre> <p>An executable function object includes values for its lexical and message environment, as well as code. This information is captured when the function constructor is evaluated at run time.</p>
global	A variable or function that is accesible from any NewtonScript code.
global data file	An NTK file named "GlobalData," in the same folder as the NTK application, that is compiled once each time you launch NTK. You can place in it NewtonScript code that you want available from any project.
immediate	A value that is stored directly rather than through an indirect reference to a heap object. Immediates are characters, integers, or Booleans. See also <b>reference</b> .
implementor	The frame in which a method is defined. See also <b>receiver</b> .
inheritance	The mechanism by which attributes (slots or data) and behaviors (methods) are made available to objects. Parent inheritance allows views of dissimilar types to share slots containing data or methods. Prototype inheritance allows a template to base its definition on that of another template or prototype.
instantiate	To make a run-time object in the NewtonScript heap from a template. Usually this term refers to the process of creating a view from a template.
layout file	A file that contains view templates laid out in NTK.
layout view	The topmost parent of all other views in a single NTK layout file.
local	A variable whose scope is the function within which it is defined. You use the <code>local</code> keyword to explicitly create a local variable within a function.

message	A symbol with a set of arguments. A message is sent using the message send syntax, <i>frame : messageName ( )</i> , where the message, <i>messageName</i> , is sent to the receiver, <i>frame</i> .
method	A function object in a frame slot that is invoked in response to a message.
NewtonScript heap	An area of RAM used by the system for dynamically allocated objects, including NewtonScript objects.
nil	A value that indicates nothing, none, no, or anything negative or empty. It is similar to <code>(void*)0</code> in C. The value <i>nil</i> represents “false” in boolean expressions; any other value represents “true.”
object	A typed piece of data that can be an immediate, array, frame, or binary object. In NewtonScript, only frame objects can hold methods and receive messages.
object stream file	See <b>stream file</b> .
package	The unit in which software can be installed on and removed from the Newton. A package consists of a header, which contains the package name and other information, and one or more parts, which contain the software.
package file	A file that contains downloadable Newton software.
package store	See <b>store part</b> .
parent	A frame that is referenced through the <code>_parent</code> slot of another frame. With regard to views, a parent view encloses its child views.
part	A unit of software—either code or data—that’s created during a single NTK build of an application, book, store, or auto part. The format of the part is identified by a four-character identifier called its type or its part code.
part frame	The top-level frame that holds an application, book, or auto part.
picker	A type of view on the Newton that pops up and contains a list of items. The user can select an item by tapping it

	in the list. This type of view closes when the user taps an item in the list or taps outside of it without making a selection.
pop-up	See <b>picker</b> .
project	The collected files and specifications that NTK uses to build a package that can be downloaded and executed on the Newton.
project file	An NTK file that contains a list of files to be included in a build and the build specifications.
proto	A frame that is referenced through another frame's <code>_proto</code> slot. With regard to views, a proto is not intended to be directly instantiated—you reference the proto from a template. The system supplies several view protos, which an application can use to implement user interface elements such as buttons, input fields, and so on.
receiver	The frame that was sent a message. The receiver for the invocation of a function object is accessible through the pseudo-variable <code>self</code> . See also <b>implementor</b> .
reference	A value that indirectly refers to an array, frame, or binary object. See also <b>immediate</b> .
resource	Raw data—usually bitmaps or sounds—stored on the development system and incorporated into a Newton application during the project build.
resource file	A file that contains Macintosh-style resources, to be used during an NTK project build.
root view	The topmost parent view in the view hierarchy. All other views descend from the root view.
self	A pseudo-variable that is set to the current receiver.
siblings	Child frames that have the same parent frame.
slot	An element of a frame or array that can hold an immediate or reference.
soup	A persistently stored object that contains a series of frames called entries. Like a database, a soup

	has indexes that can be used to access entries in a sorted order.
stack activation record	A frame on the function-call stack that describes a function that has not yet completed execution. A stack activation record contains a pointer to the next instruction that's to be executed; the function's receiver and implementor, if any; and the function's parameters, temporary variables, and named variables.
store	A physical repository that can contain soups and packages. A store is like a volume on a disk on a personal computer.
store part	A part that encapsulates a read-only store. This store may contain one or more soup objects. Store parts permit soup-like access to read-only data residing in a package. Store parts are sometimes referred to as package stores.
stream file	A file encoded in Newton Streamed Object Format (NSOF). You can use NTK to build stream files, and you can incorporate stream files into NTK projects.
template	A frame that contains the data description of an object (usually a view). A template is intended to be instantiated at run time. See also <b>proto</b> .
text file	A file that contains text to be compiled during the build.
user proto	A proto defined by an application developer, not supplied by the system.
view	The object that is instantiated at run time from a template. A view is a frame that represents a visual object on the screen. The <code>_proto</code> slot of a view references its template, which defines its characteristics.
view class	A primitive building block on which a view is based. All view protos are based directly or indirectly (through another proto) on a view class. The view class of a view is specified in the <code>viewClass</code> slot of its template or proto.



# Index

## A

---

About Newton Toolkit 9-31  
activation records 6-12, 7-6  
Add File command 9-15  
Add Window 9-15  
afterScript slots 4-54  
Align command 9-25  
Alignment command 5-9 to 5-10, 9-24  
application base view 3-6, 5-3  
Application/Book Characteristics ?? to 4-9  
application parts 4-18, 4-48  
application settings 4-6  
Apply command 5-18, 9-28  
App Preferences 4-20  
ARM machine code. See native code  
Arrange Icons 9-30  
array GL-1  
Arrow Keys Move By 4-25  
Auto Close 4-8  
Auto Download After Building Package 4-22  
Autogrid On command 9-22  
Auto Indent 4-28  
auto parts 4-18, 4-49  
Auto Remove Package 4-11  
Auto Save Before Building Package 4-22

## B

---

base view  
    application 3-6  
    layout 3-17  
beforeScript slots 4-53  
binary object GL-1  
bitmap 5  
BMP 5  
book parts 4-18, 4-49  
Boolean GL-1  
Boolean slots 5-22  
BreakLoop function 6-5, 6-11 to 6-12, 6-26  
    user modification functions 7-5, 7-13 to 7-14  
break loops 6-11 to 6-12, 6-26, 7-3 to 7-5, 7-13 to 7-15  
breakOnThrows variable 6-12, 6-21  
break points 7-3 to 7-5, 7-10 to 7-15  
browser 5-16 to 5-24  
    adding non-view objects 5-28  
    browsing templates 5-16 to 5-19

    editing templates 5-18 to 5-24, 5-28  
    preferences settings 4-25 to ??  
    searching for text 5-25 to 5-28  
Browser Preferences command 4-25 to ??  
build heap 4-23  
Build Package command 4-43 to 4-55, 9-16  
    Output Settings 4-16 to ??  
    Package Settings 4-9  
    processing templates 4-53 to 4-54  
    Project Settings 4-12  
byte code 4-43, 7-7  
    displaying 7-7 to 7-8  
    interpreter instructions 7-24 to 7-49  
    suppressing 4-15

## C

---

Cascade 9-30  
Check Global Function Calls 4-14  
child GL-2  
class GL-2  
Clear command 9-6  
Clone function 6-37  
Close command 9-3  
Command Reference 9-30  
Compile for Debugging 4-13, 4-44 to 4-45, 7-2  
Compile for Profiling 4-15  
compiler options ?? to 4-12, 4-12 to ??, 4-43 to 4-50  
compile-time functions 4-36 to 4-42  
Connect Inspector command 9-29  
constants GL-2  
    defined by NTK 4-34 to 4-36  
    defining 4-31, 4-37  
Copy command 9-5  
Copy Protected 4-11  
copyright, package 4-12  
custom parts 4-18, 4-51  
Cut command 9-5

## D

---

Debug function 3-33, 6-7 to 6-8, 6-23  
DebuggerInfo slot 7-2  
debugging 6-1 to 6-40, 7-1 to 7-49  
    break loops 6-11 to 6-13, 6-26, 7-3 to 7-5

- break points 7-3 to 7-5, 7-10 to 7-15
- displaying interpreter instructions 7-7 to 7-8, 7-21 to 7-22, 7-24 to 7-49
- examining the stack 6-12 to 6-13, 7-6, 7-16 to 7-20
- functions for 6-22 to 6-30, 7-9 to 7-22
- stepping through code 7-15 to 7-16
- trace variable 6-14, 6-21
- tracing execution flow 6-14 to 6-15
- tutorial 3-31 to 3-35
- variables for 6-21 to 6-22
- DebugHashToName package 7-2
- debug slot 4-13, 6-7
- declaring views 5-13
- DefConst function 4-38
- DefineGlobalConstant function 4-37 to 4-38
- Delete Old Package on Download 4-10
- deletion script 4-40
- Disasm function 7-7, 7-21
  - controlling display 7-24
- DisasmRange function 7-21
- Display function 6-9, 6-25
- Download Package command 9-16
  - Auto Download After Building Package 4-22
  - Delete Old Package on Download 4-10
- drawing 6-20 to 6-21
- Duplicate command 9-6
- DV function 3-34, 6-8 to 6-9, 6-23

## E

---

- EditCmds 1-3
- EnableBreakPoint function 7-11
- error messages 4-54
- Espy.fon 1-3
- evaluate slots 5-20
- Exception Printer application E-1
- exceptions, breaking for 6-11
- execution flow, tracing 6-14 to 6-15
- Exit 9-4
- Exit Break Loop button 6-12
- ExitBreakLoop function 6-12, 6-26
- Export Package to Text command 9-16
- extended debugging functions 7-2 to 7-22

## F

---

- Faster Compression 4-11
- Faster Functions 4-16
- Faster Stores 4-18
- files
  - adding to a project 4-2

- global data 4-29
- layout 4-3, 5-14
- object stream 4-5, 4-18, 4-50 to 4-51
- package 4-5
- project 4-2 to 4-3
- proto 5-16
- resource 4-4
- saving automatically 4-22
- text 4-4, 4-31 to 4-38
- Find command 5-26 to 5-27, 9-8
- Find Inherited command 5-27, 9-8
- Find Next command 5-26 to 5-27, 9-8
- font spec GL-2
- For Newton 2.0 Only 4-16
- frames GL-2
- function-call stack 7-6
- function objects 7-7, GL-3
- functions
  - BreakLoop 6-5, 6-11 to 6-12, 6-26, 7-5, 7-13 to 7-14
  - Clone 6-37
  - compile-time 4-36 to 4-42
  - Debug 3-33, 6-23
  - debugging 6-22 to 6-30, 7-9 to 7-22
  - DefineGlobalConstant 4-37 to 4-38
  - Disasm 7-7, 7-21
  - DisasmRange 7-21
  - Display 6-9, 6-25
  - DV 3-34, 6-8 to 6-9, 6-23
  - EnableBreakPoint 7-11
  - ExitBreakLoop 6-12, 6-26
  - GC 6-29
  - GetAllBreakPoints 7-12
  - GetAllNamedVars 7-19
  - GetAllTempVars 7-18
  - GetBreakPointLabel 7-13
  - GetCurrentFunction 7-17
  - GetCurrentImplementor 7-20
  - GetCurrentPC 7-17
  - GetCurrentReceiver 7-20
  - GetLayout 4-39
  - GetLocalFromStack 6-27
  - GetNamedVar 7-19
  - GetPathToSlot 7-7, 7-20
  - GetPathWhereSet 7-7, 7-21
  - GetSelfFromStack 6-27
  - GetSound11 10
  - GetTempVar 7-18
  - GloballyEnableBreakPoints 7-12
  - HasSlot 6-37
  - InstallBreakPoint 7-4, 7-11
  - IsGlobalConstant 4-39
  - Load 4-42
  - LocObj 4-46
  - NSDBreakLoopEntry 7-5, 7-14
  - NSDBreakLoopExit 7-5, 7-15

- primitive 7-40
- Print 6-9, 6-24
- QuickStackTrace 7-16
- ReadStreamFile 4-42
- RemoveAllBreakPoints 7-11
- RemoveBreakPoint 7-11
- resource-handling 8 to 10
- RunUntil 7-16
- SetBreakPointLabel 7-13
- SetCurrentPC 7-17
- SetNamedVar 7-19
- SetTempVar 7-18
- StackTrace 6-12 to 6-13, 6-27, 7-6, 7-16
- Stats 6-16, 6-28
- Step 7-15
- StepIn 7-15
- StepOut 7-15
- TrueSize 6-16 to 6-20, 6-28
- UndefineGlobalConstant 4-38
- ViewAutopsy 6-20 to 6-21, 6-30
- Where 7-18
- Write 6-9, 6-25

## G

---

- GC function 6-29
- GetAllBreakPoints function 7-12
- GetAllNamedVars function 7-19
- GetAllTempVars function 7-18
- GetBMPAsBits 9
- GetBreakPointLabel function 7-13
- GetCurrentFunction function 7-17
- GetCurrentImplementor function 7-20
- GetCurrentPC function 7-17
- GetCurrentReceiver function 7-20
- GetLayout function 4-39
- GetLocalFromStack function 6-27
- GetNamedVar function 7-19
- GetPathToSlot function 7-7, 7-20
- GetPathWhereSet function 7-7, 7-21
- GetSelfFromStack function 6-27
- GetSound11 function 10
- GetTempVar function 7-18
- GetWAVAsSamples 8
- global GL-3
- global data file 4-29
- GloballyEnableBreakPoints function 7-12
- glossary GL-1
- Grid On 4-24

## H

---

- hardware requirements 1-2
- HasSlot function 6-37
- HeapShow application E-2 to E-9
- Heaps Preferences 4-23
- home constant 4-34, 4-35

## I

---

- Icon 4-8
- Ignore Native Keyword 4-14
- immediate value GL-3
- indenting 4-28
- Index 9-30
- inheritance GL-3
- Inspector 6-2 to 6-22
  - connecting 1-6 to 1-7
- Inspector Toolbar 4-20
- installation
  - connecting a Newton to a Macintosh 1-4 to ??
  - installing NS Debug Tools on a Newton 7-2
  - installing NTK on a Macintosh 1-2 to ??
  - installing the Toolkit application on a Newton 1-4 to 1-6
  - troubleshooting 1-7 to 1-8
- InstallBreakPoint function 7-4, 7-11
- install scripts 4-32 to 4-33
- Install Toolkit App command 1-4, 9-17
- instantiation 2-2
- IsGlobalConstant function 4-39

## K

---

- kAppName constant 4-34
- kAppString constant 4-34
- kAppSymbol constant 4-35
- kDebugOn constant 4-35
- keyboard text-editing commands A-1 to A-4
- kIgnoreNativeKeyword constant 4-35
- kPackageName constant 4-35
- kProfileOn constant 4-35

## L

---

- language string, for localization 4-13, 4-35, 4-46
- lastExError variable 6-39
- lastExMessage variable 6-39
- lastEx variable 6-39

layout\_filename constant 4-36, 4-53  
 layout base view 3-17  
 layout files 4-3  
   constants and variables referencing 4-36  
   creating 3-7, 5-1  
   defined 2-3, 5-3  
   linking 3-19 to 3-21, 5-14 to 5-16  
 Layout Preferences command 4-24 to 4-25  
 Layout Size command 9-22  
 Layout Toolbar 4-20  
 layout view 5-3  
 linked subviews 3-16, 5-14 to 5-16  
   defined 5-3  
 Link Layout command 5-14, 9-2  
 LoadDataFile 10  
 Load function 4-42  
 localization frame 4-46  
 LocObj function 4-46

## M

---

main heap 4-23  
 Mark As Main Layout command 4-3  
 Mark as Main Layout command 9-17  
 masks 5-23  
 memory  
   displaying heap-use statistics E-2 to E-9  
   measuring free memory 6-16  
   measuring objects in memory 6-16 to 6-20  
   Newton memory management E-2 to E-3  
 messages GL-4  
 methods GL-4  
 Move Backward command 5-10, 9-23  
 Move Forward command 5-10, 9-23  
 Move To Back command 5-10, 9-23  
 Move To Front command 5-10, 9-23  
 MsgPad.txt 1-3

## N

---

name  
   package 4-10  
 naming views 5-13  
 native code 8-10 to 8-20  
   compiler options 4-14, 4-15, 4-43  
   functions optimized for calling from 8-13  
   marking functions for native compiling 8-11  
   profiling 8-19 to 8-20  
   suppressing 4-14  
 New Browser command 5-16, 9-29  
 New Layout command 5-1 to 5-3, 9-1

New Project command 9-15  
 New Proto Template command 5-16, 9-2  
 New Slot command 5-19 to 5-20, 9-26  
 New Text File command 9-2  
 NewtonScript heap GL-4  
 NewtonScript heap. See also memory  
 Newt Screen Shot 9-9  
 nil GL-4  
 NSDBreakLoopEntry function 7-5, 7-14  
 NSDBreakLoopExit function 7-5, 7-15  
 NS Debug Tools package 7-2  
 NSDParamFrame 7-24  
 NTK 1.0 Build Rules 4-14  
 number slots 5-22

## O

---

objects GL-4  
 object stream files 4-5, 4-18, 4-41, 4-50 to 4-51  
 Open command 9-2  
 Open Inspector command 6-4, 9-29  
 Open Layout command 5-2, 9-29  
 Open Project command 9-15  
 Output Settings command 4-16 to ??

## P

---

package files 4-5  
 packages  
   copy protecting 4-11  
   defined 2-3  
   downloading 4-10, 4-22  
   part types 4-47 to 4-50  
   version number 4-12  
 Package Settings command 4-9 to 4-12  
 Packages Preferences 4-21  
 parent GL-4  
 part frame 4-32, 4-33  
 parts 4-47 to 4-50  
   in auto-remove packages 4-50  
   specifying type 4-18 to ??  
 Paste command 9-5  
 picker GL-4  
 'PICT' resources 4-4, 6 to 8  
   application icon ?? to 4-9  
   in picture slots 5-21 to 5-23  
 picture slots 5-23  
 platform files 4-30  
   specifying 4-13  
 Platforms 1-3  
 Preview command 5-11, 9-25

- primitive functions 7-40
- Print command 9-4
- printDepth variable 6-22
- printFormat\_filename variable 4-36
- Print function 6-9, 6-24
- printLength variable 6-22
- Print One command 9-4
- Print Setup 9-4
- Process Earlier command 4-3, 9-17
- Process Later command 4-3, 9-17
- Profile Native Functions 4-15
- profiler 8-1 to 8-10
  - configuring on the development system 8-4 to 8-6
  - configuring on the Newton 8-6 to 8-7
  - marking functions for profiling 8-2 to 8-4
  - profiling native functions 8-19 to 8-21
- program counter 6-13, 7-8
- programming problems 6-32 to 6-40
  - comparing with nil 6-34 to 6-35
  - dangling frame references 6-34
  - printing in communications code 6-38
  - resizing read-only objects 6-36 to 6-37
  - setting the function value 6-34
  - setting the wrong slot value 6-32 to 6-34
  - text not drawing 6-38
  - using nil in expressions 6-36
- programming tips 6-32 to 6-40
  - accessing parent view 6-40
  - examining exceptions 6-39
  - maintaining view state 6-39
- project file 4-2 to 4-3
- projects
  - defined 2-3
  - managing 4-1 to 4-6
  - Project Settings 4-12 to 4-15
  - project window 4-2
- Project Settings command 4-12 to 4-15
- protos 5-3, GL-5
  - previewing 5-11
  - user protos 5-16

## Q

---

QuickStackTrace function 7-16

## R

---

- read-only objects, copying 6-37
- ReadStreamFile function 4-42
- receiver GL-5
- Recent File 9-4

- rectangle slots 5-23
- Redo 9-5
- references GL-5
- RemoveAllBreakPoints function 7-11
- RemoveBreakPoint function 7-11
- Remove File command 9-15
- remove frame 4-33, 4-34
- remove scripts 4-33 to 4-34
- Rename Slot command 9-27
- REP loop 7-3
- resource files 4-4
  - adding to a project 5
  - opening and closing 5 to ??
- resources ?? to 11
  - application icon ?? to 4-9
  - 'PICT' 5-23, 6 to 8
  - retrieving 8 to 10
  - 'SND' 8
- Result 4-19
- Revert command 9-4, 9-28
- root view GL-5
- RunUntil function 7-16

## S

---

- Save All command 9-3
- Save As command 9-3
- Save command 9-3
- Screen Shot 9-9
- script slots 5-21
- Search command 5-25 to 5-26, 9-7
- Select All command 9-6
- Select Hierarchy command 9-6
- Select in Layout command 9-7
- self GL-5
- SetBreakPointLabel function 7-13
- SetCurrentPC function 7-17
- Set Default Window Position 9-30
- Set Grid command 9-22
- SetLocalizationFrame function 4-46 to 4-47
- SetNamedVar function 7-19
- SetTempVar function 7-18
- Settings 9-17
- Shift Left command 9-6
- Shift Right command 9-6
- Show Slot Values command 9-28
- Size 4-24
- slot
  - global GL-3
- slots
  - creating 5-19 to 5-23
  - displaying in browser 4-27
  - editing 5-20 to 5-24

- in stack trace 6-12
- searching for 5-25 to 5-28
- slot types 5-20 to 5-23
- Slots By Name command 9-28
- Slots By Type command 9-28
- Snarf application E-1
- 'SND' resources 8
- software requirements 1-2
- sound 5
- stack activation records 6-12, 7-6
- stack level 6-12, 7-16
- stacks 7-6
  - functions for manipulating 7-16 to 7-20
  - stack trace 6-12 to 6-13
- StackTrace function 6-12 to 6-13, 6-27, 7-6, 7-16
- Standard Toolbar 4-20
- Stats function 6-16, 6-28
- Step function 7-15
- StepIn function 7-15
- StepOut function 7-15
- store parts 4-18, 4-50, GL-6
- stores GL-6
- stream files 4-5, 4-18, 4-41, 4-50 to 4-51
- Suppress Byte Code 4-15
- symbol, application or book 4-7

## T

---

- tabs, setting 4-28
- Template Info command 3-9, 5-13 to 5-14, 9-25
- templates
  - defined 2-2
  - displaying in browser 4-26
  - editing 3-11 to 3-13, 5-16 to 5-24
  - processing 4-53 to 4-54
  - searching for 5-25 to 5-28
- Templates By Hierarchy command 9-27
- Templates By Type command 9-27
- text
  - editing 5-23 to 5-24, A-1 to ??
  - searching for 5-25 to 5-27
  - setting display characteristics 4-28
  - setting tabs 4-28
- text files 4-4, 4-31 to 4-38
- text slots 5-21
- Text Views Preferences 4-27
- Tighter Object Packing 4-16
- Tile 9-30
- Toolkit application 1-4 to 1-6
- Toolkit.pkg 1-3
- Toolkit Preferences command 4-19 to ??, 9-9
- Tooltips 4-21
- Trace Off button 6-15

- trace variable 6-14, 6-21
- troubleshooting 1-7 to 1-8
- TrueSize function 6-16 to 6-20, 6-28

## U

---

- UndefineGlobalConstant function 4-38
- Undo command 9-5
- Update File command 9-16
- Use compression 4-11
- user proto templates
  - creating 5-16
  - example 3-23 to 3-30
- Using Help 9-30

## V

---

- variables defined by NTK 4-34 to 4-36
- version number, package 4-12
- vFlags application E-1
- ViewAutopsy function 6-20 to 6-21, 6-30
- viewBounds slot fields D-2
- view classes 5-3, GL-6
- view frames 2-2
- viewFrontKey 6-9, 6-24
- viewFrontMost 6-9, 6-24
- viewFrontMostApp 6-9, 6-24
- views
  - aligning 5-8 to 5-10
  - declaring 5-13
  - defined 2-1
  - displaying hierarchy 6-8 to 6-9, 6-23
  - displaying in browser 4-26
  - drawing 3-6 to 3-9, 5-4 to 5-6
  - moving 5-8
  - naming 3-9, 5-13
  - ordering 5-10
  - previewing 5-11
  - resizing 5-7
  - root GL-5

## W

---

- WAV 5
- Where function 7-18
- Write function 6-9, 6-25

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro 630 printer. Final page negatives were output directly from the text and graphics files. Line art was created using Adobe<sup>™</sup> Illustrator. PostScript<sup>™</sup>, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino<sup>®</sup> and display type is Helvetica<sup>®</sup>. Bullets are ITC Zapf Dingbats<sup>®</sup>. Some elements, such as program listings, are set in Apple Courier.

LEAD WRITER

Aner J. Menendez

WRITERS

Aner J. Menendez, Norberto Menendez

PROJECT LEADER

Christopher Bey

ILLUSTRATORS

Aner J. Menendez, Norberto Menendez

EDITOR

David Schneider

PROJECT MANAGER

Gerry Kane

Special thanks to Andy Atkins, Peter Canning, Jerome Coonen, Bob Ebert, Mike Engber, Sandy McEntee, David Fedor, Sue Luttner, Ray Marshall, Jeff Piazza, Uri Rabin, Keith Rollins, Walter Smith, Michael Tibbott, Gregory Toto.

# THE APPLE PUBLISHING SYSTEM

---